Python
中文官方文档
2.7 & 3.4

wizardforcel

Published
with GitBook

目錄

Python 2.7 & 3.4 中文官方文档

来源：python.usyiyi.cn

整理：飞龙

Python 2 教程

Python是一门简单易学，功能强大的编程语言。它具有高效的高级数据结构和简单而有效的面向对象编程方法。Python优雅的语法和动态类型以及其解释性的性质，使它在许多领域和大多数平台成为编写脚本和快速应用程序开发的理想语言。

从Python网站http://www.python.org/可以免费获得所有主要平台的源代码或二进制形式的Python解释器和广泛的标准库，并且可以自由地分发。该网站还包含许多免费的第三方Python 模块、程序、工具以及附加文档的发布包和链接。

Python解释器可以用C或C++（或可从C中调用的其他语言）中实现的新的函数和数据类型轻松扩展。Python也适合作为可定制应用程序的一种扩展语言。

本教程非正式向读者介绍Python语言及其体系的基本概念和功能。手边有个Python解释器来随手实验很有帮助，但所有示例都相对独立，所以本教程也可以离线阅读。

对于标准对象和模块的说明，请参阅*Python标准库*。*Python语言参考*给出了Python语言更正式的定义。要编写C或C++的扩展，请阅读*扩展和嵌入Python解释器*与*Python/C API参考手册*。也有几本书深度地介绍了Python 。

本教程不会尝试全面并涵盖每一个单独特性，甚至每一个常用的特性。相反，它介绍了许多Python最值得注意的特点，并会给你一个很好的语言的口味和风格。读完它之后，你将能够阅读和编写Python的模块和程序，并可以准备好更多地了解*Python标准库*中描述的各种Python库模块。

词汇表也值得浏览一下。

1. 吊吊你的胃口

如果你要用计算机做很多工作，最终你发现是有一些您希望自动执行的任务。例如，你可能希望对大量的文本的文件执行搜索和替换，或以复杂的方式重命名并重新排列一堆照片文件。也许你想写一个小的自定义数据库，或一个专门的GUI应用程序或一个简单的游戏。

如果你是一个专业的软件开发人员，您可能必须使用几个C/C++/Java库，但发现通常的编写/编译/测试/重新编译周期太慢。也许你要写这样的库中的测试套件，然后发现编写测试代码是很乏味的工作。或也许您编写了一个程序，它可以使用一种扩展语言，但你不想为您的应用程序设计与实现一个完整的新语言。

Python正是这样为你准备的语言。

你可以为其中一些任务写一个Unix shell脚本或Windows批处理文件，但是shell脚本最适合移动文件和更改文本数据，不适合用于GUI应用程序或游戏。你可以写一个C/C++/Java程序，但是甚至程序的第一个初稿都可能花费大量的开发时间。Python更简单易用，可用于Windows、Mac OS X和Unix操作系统，并将帮助您更快地完成工作。

Python使用很简单，但它是一个真正的编程语言，比shell脚本或批处理文件对于大型的程序提供更多的结构和支持。另一方面，Python还提供比C更多的错误检查，并且，作为一种高级语言，它有内置的高级数据类型，比如灵活的数组和字典。因为其更加一般的数据类型，Python比Awk甚至Perl适用于很多更大的问题领域，而且在Python中很多事情至少和那些语言一样容易。

Python允许您将您的程序拆分成可以在其它Python程序中重复使用的模块。它拥有大量的标准模块，你可以将其用作你的程序的基础 — 或者作为学习Python编程的示例。这些模块提供诸如文件I/O、系统调用、套接字和甚至用户图形界面接口，例如Tk。

Python是一门解释性的语言，因为没有编译和链接，它可以节省你程序开发过程中的大量时间。Python解释器可以交互地使用，这使得试验Python语言的特性、编写用后即扔的程序或在自底向上的程序开发中测试功能非常容易。它也是一个方便的桌面计算器。

Python使程序编写起来能够紧凑和可读。编写的Python程序通常比等价的C、C++或Java程序短很多，原因有几个：

- 高级数据类型允许您在单个语句中来表达复杂的操作；
- 语句分组是通过缩进，而不是开始和结束的括号；
- 任何变量或参数的声明不是必要的。

Python是可扩展的：如果你知道如何用C编程，那么将很容易添加一个新的内置函数或模块到解释器中，要么为了以最快的速度执行关键的操作，要么为了将Python程序与只有二进制形式的库（如特定供应商提供的图形库）链接起来。一旦你真的着迷，你可以把Python解释器链接到C编写的应用程序中，并把它当作那个程序的扩展或命令行语言。

顺便说一句，Python语言的名字来自于BBC的“Monty Python’s Flying Circus”节目，与爬行动物无关。在文档中引用Monty Python短剧不仅可以，并且鼓励！

既然现在你们都为Python感到兴奋，你们一定会想更加详细地研究它。学习一门语言最好的方法就是使用它，本教程推荐你边读边使用Python解释器练习。

在下一章中，我们将解释Python解释器的用法。这是很简单的一件事情，但它有助于试验后面的例子

本教程的其余部分通过实例介绍Python语言和体系的各种特性，以简单的表达式、语句和数据类型开始，然后是函数和模块，最后讲述高级概念，如异常和用户自定义的类。

2. Python 解释器

2.1 调用解释器

在可用的机器上，Python解释器通常安装成/usr/local/bin/python；请将/usr/local/bin放在您的Unix shell搜索路径，以使得可以通过在shell中键入命令

```
python
```

来启动它。由于解释器放置的目录是一个安装选项，其它地方也是可能的；请与您本地的Python专家或系统管理员联系。（例如，/usr/local/python是另外一个常见的位置。）

在Windows机器上，Python的安装通常放在C:\Python27，当然你可以在运行安装程序时进行更改。你可以在一个DOS窗口的命令提示符下键入以下命令来把这个目录添加到路径中：

```
set path=%path%;C:\python27
```

主提示符下键入文件结束字符（Unix上是Control-D、Windows上是Control-Z）会导致解释器以0退出状态退出。如果无法正常工作，您可以通过键入以下命令退出解释器：quit()。

解释器的行编辑功能通常不是很复杂。在Unix上，不管是谁安装的，解释器可能已启用对GNU readline库的支持，该库添加了更详细的交互式编辑和历史记录功能。检查是否支持命令行编辑的最快的方式也许是对你的第一个Python提示符键入Control-P。如果它发出蜂鸣声，则有命令行编辑；请参阅附录交互式输入编辑和历史替代的有关快捷键的介绍。如果什么都没发生，或者显示^P，则命令行编辑不可用；你就只能够使用退格键删除当前行中的字符。

解释器有些像Unix shell：当调用时使用连接到一个tty设备作为标准输入，它交互地读取并执行命令；当用文件名参数或文件作为标准输入调用，它将读取并执行该文件中的脚本。

第二种启动解释器的方式是python-ccommand[arg]...，它会执行*command*中的语句，类似于shell的*-c*选项。因为Python语句经常包含空格或其他shell特殊字符，通常建议把全部*command*放在单引号里。

有些Python模块也是可执行的脚本。这些模块可以使用python-mmodule[arg]...直接调用，这和在命令行输入完整的路径名执行*module*的源文件是一样的。

有时使用一个脚本文件，能够在运行该脚本之后进入交互模式非常有用。这可以通过在脚本前面加上*-i*选项实现。

2.1.1. 参数传递

调用解释器时，脚本名称和其他参数被转换成一个字符串列表并赋值给sys模块中的argv变量。你可以通过importsys访问此列表。列表的长度是至少是1；如果没有给出脚本和参数，sys.argv[0]是一个空字符串。当使用-c*command*时，sys.argv[0]被设置为'-c'。当使用-m*module*时，sys.argv[0]被设定为指定模块的全名。-c*command*或-m*module*后面的选项不会被Python解释器的选项处理机制解析，而是被保存在sys.argv中，供命令或模块使用。

2.1.2. 交互模式

当命令从tty读取时，就说解释器在交互模式下。这种模式下解释器以主提示符提示下一个命令，主提示符通常为三个大于号（>>>）；对于续行解释器以从提示符提示，默认为三个点（...）。在第一个提示符之前，解释器会打印出一条欢迎信息声明它的版本号和授权公告：

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行结构时需要续行。作为一个例子，看看这个if语句：

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2. 解释器及其环境

2.2.1. 错误处理

错误发生时，解释器会打印错误信息和堆栈跟踪信息。在交互模式下，它会返回到主提示符；当输入来自一个文件，它会打印堆栈跟踪信息，然后以非零退出状态退出。（由try语句中的except子句处理的异常不是这方面的错误）。有些错误是致命的并导致非零状态退出；这通常由于内部不一致和某些情况下的内存不足导致。所有错误消息都写入标准错误流；执行命令的普通输出写入标准输出。

在主提示符或从提示符后输入中断符(通常为Control-C或DEL)可以取消输入，并返回到主提示符。[1]命令执行过程中输入中断符将引发KeyboardInterrupt异常，它可以被try语句截获。

2.2.2. 可执行的Python脚本

在类BSD的Unix系统上，可以将Python脚本变成可直接执行的，就像shell脚本一样，通过放置一行

```
#! /usr/bin/env python
```

（假定解释器在用户的PATH中）在脚本的开始并且给文件可执行的模式。#!必须是文件最开始的两个字符。在一些平台上，第一行必须以一个Unix风格的行结束符（'\n'）结束，不能是Windows的行结束符（'\r\n'）。注意，字符'#'在Python中用于起始一个注释。

通过**chmod**命令可以给予脚本可执行的模式或权限：

```
$ chmod +x myscript.py
```

在Windows系统上，没有"可执行模式"的概念。Python安装程序会自动将.py文件与python.exe关联，双击Python文件将以脚本的方式运行它。扩展名也可以是.pyw，在这种情况下，通常出现的控制台窗口不会再显示了。

2.2.3. 源程序的编码

在Python源文件中可以使用非ASCII编码。最好的方法是在#!行的后面再增加一行特殊的注释来定义源文件的编码：

```
# -*- coding: encoding -*-
```

通过此声明，源文件中的所有字符将被视为由*encoding*编码，并且可以直接写由选中的编码方式编码的Unicode字符串字面量。在Python库参考手册的codecs小节中，可以找到所有可能的编码方式列表。

例如，若要写入包含欧元货币符号的 Unicode 字面量，可以使用ISO-8859-15 编码，其欧元符号的值为164 。此脚本中，以 ISO-8859-15 编码，保存时将打印的值 8364 (Unicode 代码点相应的欧元符号），然后退出：

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

如果你的编辑器支持保存为带有UTF-8字节顺序标记(也叫做BOM) 的UTF-8格式的文件，你可以使用这种功能而不用编码声明。IDLE如果设置了Options/General/Default Source Encoding/UTF-8 也支持此功能。注意，这种标记方法在旧的Python 版本中（2.2 及更早）是不能识别的，同样也不能被能够处理#!（只在Unix系统上使用）行的操作系统识别。

通过使用 UTF-8 编码（无论是BOM方式或者是编码声明方式），世界上大多数语言的字符可以在字符串字面量和注释中同时使用。在标识符中使用非 ASCII 字符是不支持的。若要正确显示所有这些字符，您的编辑器必须认识该文件是 UTF-8 编码，并且它必须使用支持文件中所有字符的字体。

2.2.4. 交互式启动文件

当您以交互方式使用Python时，让解释器在每次启动时执行一些标准命令会变得非常方便。你可以通过设置环境变量PYTHONSTARTUP为包含你的启动命令的文件的名字。这类似于Unix shell的.profile功能。

这个文件只会在交互式会话时读取，当 Python 从脚本中读取命令时不会读取，当/dev/tty 在命令中明确指明时也不会读取 （尽管这种方式很像是交互方式）。它和交互式命令在相同的命名空间中执行，所以在交互式会话中，由它定义或引用的一切可以在解释器中不受限制地使用。您还可以在此文件中更改sys.ps1和sys.ps2 的提示符。

如果您想要从当前目录读取额外的启动文件，你可以在全局启动文件中使用这样的代码 ifos.path.isfile('.pythonrc.py'):execfile('.pythonrc.py')。如果你想要在脚本中使用启动文件，必须要在脚本中明确地写出：

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

2.2.5. 自定义模块

Python提供了两个钩子来定制化它：sitecustomize和usercustomize。要查看它如何工作，你首先需要找到你的用户site-packages目录。启动Python并运行下面的代码：

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

现在你可以在此目录下创建名为usercustomize.py的文件，并把任何你想要的东西放在里面。它将影响每个Python调用，除非启动时用*-s*选项来禁用自动导入。

sitecustomize的工作方式相同，但通常是由计算机的管理员在全局site-packages目录中创建，并在usercustomize之前导入。更多详细信息请参阅site模块的文档。

脚注

| [1] | GNU Readline库的一个问题可能导致它不会发生。 | |-----|-----|

3. Python简介

以下的示例中，输入和输出通过是否存在提示符（>>>和...）来区分：如果要重复该示例，你必须在提示符出现后，输入提示符后面的所有内容；没有以提示符开头的行是解释器的输出。注意示例中出现从提示符意味着你一定要在最后加上一个空行；这用于结束一个多行命令。

本手册中的很多示例，甚至在交互方式下输入的示例，都带有注释。Python中的注释以哈希字符#开始，直至实际的行尾。注释可以从行首开始，也可以跟在空白或代码之后，但不能包含在字符串字面量中。字符串字面量中的#字符仅仅表示#。因为注释只是为了解释代码并且不会被Python解释器解释，所以敲入示例的时候可以忽略它们。

例如：

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. 用Python作为计算器

让我们尝试一些简单的Python命令。启动解释器然后等待主提示符 >>>。（应该不需要很久。）

3.1.1. 数字

解释器可作为一个简单的计算器：你可以向它输入一个表达式，它将返回其结果。表达式语法很直白：运算符+、-、*和/的用法就和其它大部分语言一样（例如Pascal或C）；括号(())可以用来分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

整数(例如2,4,20)的类型是int，带有小数部分的数字(例如5.0, 1.6)的类型是float。在本教程的后面我们会看到更多关于数字类型的内容。

除法(/)返回的类型取决于它的操作数。如果两个操作数都是int，将采用*floor除法(floor division)*并返回一个int。如果两个操作数中有一个是float，将采用传统的除法并返回一个float。还提供//运算符用于floor division 而无论操作数是什么类型。余数可以用%操作符计算：

```
>>> 17 / 3  # int / int -> int
5
>>> 17 / 3.0  # int / float -> float
5.666666666666667
>>> 17 // 3.0  # explicit floor division discards the fractional part
5.0
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

通过Python，还可以使用**运算符计算幂乘方[1]：

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

等号（=）用于给变量赋值。赋值之后，在下一个提示符之前不会有任何结果显示：

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

如果变量没有“定义”（赋值），使用的时候将会报错：

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python中完全支持浮点数；整数和浮点数的混合计算中，整数会被转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

在交互模式下，最近一次表达式的值被赋给变量_。这意味着把Python当做桌面计算器使用的时候，可以方便的进行连续计算，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

用户应该将这个变量视为只读的。不要试图去给它赋值 — 你将会创建出一个独立的同名局部变量，并且屏蔽了内置变量的魔术效果。

除了int和float，Python还支持其它数字类型，例如DecimalandFraction。Python还内建支持复数，使用后缀j或J表示虚数部分（例如3+5j）。

3.1.2. 字符串

除了数值，Python 还可以操作字符串，可以用几种方法来表示。它们可以用单引号('...')或双引号("...")括起来，效果是一样的[2]。\可以用来转义引号。

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

在交互式解释器中，输出的字符串会用引号引起来，特殊字符会用反斜杠转义。虽然可能和输入看上去不太一样，但是两个字符串是相等的。如果字符串中只有单引号而没有双引号，就用双引号引用，否则用单引号引用。

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print '"Isn\'t," she said.'
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print s  # with print, \n produces a new line
First line.
Second line.
```

如果你前面带有\的字符被当作特殊字符，你可以使用原始字符串，方法是在第一个引号前面加上一个r:

```
>>> print 'C:\some\name'  # here \n means newline!
C:\some
ame
>>> print r'C:\some\name'  # note the r before the quote
C:\some\name
```

字符串可以跨多行。一种方法是使用三引号："""..."""或者'''...'''。行尾换行符会被自动包含到字符串中，但是可以在行尾加上\来避免这个行为。下面的示例：

```
print """\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

将生成以下输出（注意，没有开始的第一行）：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

字符串可以用+操作符连接，也可以用*操作符重复多次：

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

相邻的两个或多个字符串字面量（用引号引起来的）会自动连接。

```
>>> 'Py' 'thon'
'Python'
```

然而这种方式只对两个字面量有效，变量或者表达式是不行的。

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

如果你想连接多个变量或者连接一个变量和一个字面量，使用+：

```
>>> prefix + 'thon'
'Python'
```

这个功能在你想切分很长的字符串的时候特别有用：

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

字符串可以索引，第一个字符的索引值为0。Python没有单独的字符类型；一个字符就是一个简单的长度为1的字符串。

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

索引也可以是负值，此时从右侧开始计数：

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

注意，因为-0和0是一样的，负的索引从-1开始。

除了索引，还支持切片。索引用于获得单个字符，切片让你获得一个子字符串。

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

注意，包含起始的字符，不包含末尾的字符。这使得s[:i]+s[i:]永远等于s：

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

切片的索引有非常有用的默认值；省略的第一个索引默认为零，省略的第二个索引默认为切片的字符串的大小。

```
>>> word[:2]  # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]  # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

有个方法可以记住切片的工作方式，把索引当做字符之间的点，第一个字符的左边是0。含有*n*个字符的字符串的最后一个字符的右边是索引*n*，例如：

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行给出了字符串中0..6各索引的位置；第二行给出了相应的负索引。从*i*到*j*的切片由*i*和*j*之间的所有字符组成。

对于非负索引，如果上下都在边界内，切片长度就是两个索引之差。例如，word[1:3] 的长度是 2。

试图使用太大的索引会导致错误：

```
>>> word[42]  # the word only has 7 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

但是，当用于切片时，超出范围的切片索引会被优雅地处理：

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python字符串不可以被更改 — 它们是不可变的。因此，赋值给字符串索引的位置会导致错误：

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

如果你需要一个不同的字符串，你应该创建一个新的：

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

内置函数len()返回字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

请参阅

序列类型 — *str, unicode, list, tuple, bytearray, buffer, xrange*字符串和下节描述的Unicode字符串是序列类型的例子，它们支持这种类型共同的操作。字符串方法字符串和Unicode字符串都支持大量的方法用于基本的转换和查找。字符串格式化这里描述了使用str.format()进行字符串格式化的信息。字符串格式化操作这里描述了旧式的字符串格式化操作，它们在字符串和Unicode字符串是%操作符的左操作数时调用。

3.1.3. Unicode 字符串

从Python2.0开始，程序员们有了一个新的用来存储文本数据的类型：Unicode对象。它可以用来存储和处理Unicode数据(见http://www.unicode.org/)，并与现有的字符串对象有良好的集成，必要时提供自动转换。

Unicode 的优点在于为现代和古代的每一种文字的每一个字符提供了统一的序号。以前，脚本只有256个可用的字符编码。通常，文本被绑定到映射字符编码的代码页上。这带来很多麻烦，尤其是软件国际化（通常写成i18n — 'i' + 18个字符 + 'n'）。Unicode 为所有脚本定义一个代码页，从而解决了这些问题。

在Python中创建Unicode字符串和创建普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前面小写的'u'表示创建一个Unicode字符串。如果你想要在字符串中包含特殊字符，你可以通过使用Python的*Unicode*转义编码。下面的示例演示如何使用：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

转义序列u0020表示在给定位置插入序号值为0x0020（空格字符）的Unicode字符。

其他字符就像 Unicode 编码一样被直接解释为对应的编码值。如果你有使用在许多西方国家使用的标准Latin-1编码的字符串，你会发现编码小于256的Unicode字符和在Latin-1编码中的一样。

和普通字符串一样，Unicode字符串也有raw模式。要使用*Raw-Unicode-Escape*编码，必须在引号的前面加上'ur'。只有在小写的'u'前面有奇数个反斜杠，才会用上面的uXXXX 转换

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

当你需要输入很多反斜杠时，raw 模式非常有用，这在正则表达式中几乎是必须的。

除了这些标准的编码，Python提供了基于已知编码来创建Unicode字符串的一整套方法。

内置函数unicode()提供对所有已注册的Unicode编解码器（编码和解码）的访问。这些编解码器可以转换的比较有名的编码有*Latin-1*、*ASCII*、*UTF-8*和*UTF-16*。后两个是可变长度编码，它们存储每个Unicode字符在一个或多个字节中。默认编码通常设置为ASCII，此编码接受0到127这个范围的编码，否则报错。当打印、向文件写入、或者用str()转换一个Unicode字符串时，转换将使用默认编码。

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in
```

Unicode对象提供encode()方法将Unicode字符串转换为使用指定编码的8位字符串，它接收一个编码名称作为参数。编码名应该小写。

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

如果有一个已知编码的数据，希望从它生成一个Unicode字符串，你可以使用unicode()函数并以编码名作为第二个参数。

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4. 列表

Python有几个复合数据类型，用来组合其他的值。最有用的是列表，可以写成中括号中的一列用逗号分隔的值。列表可以包含不同类型的元素，但是通常所有的元素都具有相同的类型。

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

和字符串（以及其它所有内建的序列类型）一样，列表可以索引和切片：

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

所有的切片操作都会返回一个包含请求的元素的新列表。这意味着下面的切片操作返回列表一个新的（浅）拷贝副本。

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表也支持连接这样的操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

与字符串的不可变特性不同，列表是可变的类型，例如可以改变它们的内容：

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

你还可以使用append()方法（后面我们会看到更多关于方法的内容）在列表的末尾添加新的元素：

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

给切片赋值也是可以的，此操作甚至可以改变列表的大小或者清空它:：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

内置函数len()也适用于列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

列表可以嵌套（创建包含其他列表的列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. 编程第一步

当然，我们可以将Python用于比计算2加2更复杂的任务。例如，我们可以写一个生成斐波那契初始子序列的程序，如下所示：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

本示例介绍了几种新功能。

- 第一行包括了一个多重赋值：变量a和b同时获得新的值0和1。最后一行又这样使用了一次，说明等号右边的表达式在赋值之前首先被完全解析。右侧表达式是从左到右计算的。
- 只要条件（这里是 b<10）为真，[while](#)循环反复执行。在Python中，和C一样，任何非零整数值都为真；零为假。循环条件也可以是一个字符串或者列表，实际上可以是任何序列；长度不为零的序列为真，空序列为假。示例中使用的测试是一个简单的比较。

标准比较运算符与 C="" 的写法一样：< （小于），="">（大于），==（等于)，<=（小于或等于），>=（大于或等于）和!=（不等于）。

- 循环体是缩进 的：缩进是Python分组语句的方式。交互式输入时，你必须为每个缩进的行输入一个 tab 或（多个）空格。实践中你会用文本编辑器来编写复杂的 Python 程序；所有说得过去的文本编辑器都有自动缩进的功能。交互式输入复合语句时，最后必须在跟随一个空行来表示结束（因为解析器无法猜测你什么时候已经输入最后一行）。注意基本块内的每一行必须按相同的量缩进。
- print语句输出传给它的表达式的值。与仅仅输出你想输出的表达式不同（就像我们在前面计算器的例子中所做的），它可以输出多个表达式和字符串。打印出来的字符串不包含引号，项目之间会插入一个空格，所以你可以设置漂亮的格式，像这样：

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

尾部的逗号可以避免输出换行符：

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

注意，如果最后一行没有结束，解释器会插入一个新行（在打印下一个提示符之前）。

脚注

[1]	因为的优先级高于-，所以-32将解释为-(32)且结果为-9。为了避免这点并得到9，你可以使用(-3)2。

| [2] | 与其它语言不同，特殊字符例如\n在单引号('...')和双引号("...")中具有相同的含义。两者唯一的区别是在单引号中，你不需要转义"（但你必须转义\'），反之亦然。 | |-----|-----|

4. 控制流

除了前面介绍的while语句，Python也有其它语言常见的流程控制语句，但是稍有不一样。

4.1. if语句

也许最知名的语句类型是if语句。例如：

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

可以有零个或多个elif部分，else部分是可选的。关键字'elif'是'else if'的简写，可以有效避免过深的缩进。if...elif...elif...序列用于替代其它语言中的switch或case语句。

4.2. for语句

Python中的for语句和你可能熟悉的C或Pascal中的有点不同。和常见的依据一个等差数列迭代（如 Pascal），或让用户能够自定义迭代步骤和停止条件（如 C）不一样，Python的for语句按照元素出现的顺序迭代任何序列（列表或字符串）。例如（没有双关意）：

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

如果要在循环内修改正在迭代的序列（例如，复制所选的项目），建议首先制作副本。迭代序列不会隐式地创建副本。使用切片就可以很容易地做到：

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3. range()函数

如果你确实需要遍历一个数字序列，内置函数range()非常方便。它将生成包含算术数列的列表：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

给定的终点永远不会在生成的列表中；range(10)生成一个包含10个值的链表，索引的值和对应元素的值相等。也可以让 range 函数从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为‘步长’）：

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

若要依据索引迭代序列，你可以结合使用range()和len()，如下所示：

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

然而，在大部分情况下使用enumerate()函数会更加方便，请参见循环的技巧。

4.4. break和continue语句，以及循环中else子句

break语句和C中的类似，用于跳出最近的for或while循环。

循环语句可以有一个else子句；当（for）循环迭代完整个列表或（while）循环条件变为假，而非由break语句终止时，它会执行。下面循环搜索质数的代码例示了这一点：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

（是的，这是正确的代码。看仔细：else子句属于for循环，不属于if语句。）

与循环一起使用时，else子句与try语句的else子句比与if语句的具有更多的共同点：try语句的else子句在未出现异常时运行，循环的else子句在未出现break时运行。更多关于try语句和异常的内容，请参见处理异常。

continue语句，也是从C语言借来的，表示继续下一次迭代：

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5. pass语句

pass语句什么也不做。它用于语法上必须要有一条语句，但程序什么也不需要做的场合。例如：

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

它通常用于创建最小的类：

```
>>> class MyEmptyClass:
...     pass
...
```

另一个使用pass的地方是编写新代码时作为函数体或控制体的占位符，这让你在更抽象层次上思考。pass语句将被默默地忽略：

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6.定义函数

我们可以创建一个生成任意上界菲波那契数列的函数：

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字def引入函数的定义。其后必须跟有函数名和以括号标明的形式参数列表。组成函数体的语句从下一行开始，且必须缩进。

函数体的第一行可以是一个可选的字符串文本；此字符串是该函数的文档字符串，或称为*docstring*。（更多关于 docstrings 的内容可以在 文档字符串一节中找到。）有工具使用 docstrings 自动生成在线的或可打印的文档，或者让用户在代码中交互浏览；在您编写的代码中包含 docstrings 是很好的做法，所以让它成为习惯吧。

执行一个函数会引入一个用于函数的局部变量的新符号表。更确切地说，函数中的所有的赋值都是将值存储在局部符号表；而变量引用首先查找局部符号表，然后是上层函数的局部符号表，然后是全局符号表，最后是内置名字表。因此，在函数内部全局变量不能直接赋值（除非在一个global语句中命名），虽然可以引用它们。

函数调用的实际参数在函数被调用时引入被调函数的局部符号表；因此，参数的传递使用传值调用（这里的值始终是对象的引用，不是对象的值）。[1]一个函数调用另一个函数时，会为该调用创建一个新的局部符号表。

函数定义会在当前符号表内引入函数名。函数名对应值的类型是解释器可识别的用户自定义函数。此值可以分配给另一个名称，然后也可作为函数。这是通用的重命名机制：

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

如果你使用过其他语言，你可能会反对说：fib不是一个函数，而是一个过程(子程序)，因为它并不返回任何值。事实上，没有return语句的函数也返回一个值，尽管是一个很无聊的值。此值被称为None（它是一个内置的名称）。如果 None只是唯一的输出，解释器通常不会打印出来。如果你真的想看到这个值，可以使用print 语句：

```
>>> fib(0)
>>> print fib(0)
None
```

写一个函数返回菲波那契数列的列表，而不是打印出来，非常简单：

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

此示例中，像往常一样，演示了一些新的 Python 功能：

- return语句从函数中返回一个值。不带表达式参数的return返回None。函数直接结束后也返回None。
- 语句result.append(a)调用列表对象result 的一个方法。方法是'隶属于'某个对象的函数，被命名成obj.methodname的形式，其中obj是某个对象（或是一个表达式），methodname是由对象类型定义的方法的名称。不同类型定义了不同的方法。不同类型的方法可能具有相同的名称，而不会引起歧义。（也可以使用 *class* 定义你自己的对象类型

和方法，请参见类）本例中所示的append()方法是列表对象定义的。它在列表的末尾添加一个新的元素。在本例中它等同于result=result+[a]，但效率更高。

4.7.更多关于定义函数

可以定义具有可变数目的参数的函数。有三种形式，也可以结合使用。

4.7.1.默认参数值

最有用的形式是指定一个或多个参数的默认值。这种方法创建的函数被调用时，可以带有比定义的要少的参数。例如：

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

这个函数可以通过几种方式调用：

- 只给出强制参数： ask_ok (' Doyoureallywanttoquit?')
- 给出一个可选的参数： ask_ok ('OKtooverwritethefile?',2）
- 或者给出所有的参数： ask_ok ('OKtooverwritethefile?',2,'Comeon,onlyyesorno!')

此示例还引入了in关键字。它测试一个序列是否包含特定的值。

默认值在定义域中的函数定义的时候计算，例如：

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

将打印5。

重要的警告：默认值只计算一次。这使得默认值是列表、字典或大部分类的实例时会有所不同。例如，下面的函数在后续调用过程中会累积传给它的参数：

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

这将会打印

```
[1]
[1, 2]
[1, 2, 3]
```

如果你不想默认值在随后的调用中共享，可以像这样编写函数：

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2. 关键字参数

函数也可以通过kwarg=value形式的关键字参数调用。例如，下面的函数：

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

接受一个必选参数（voltage）和三个可选参数（state,action和type）。可以用下列任意一种方式调用这个函数：

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

但下面的所有调用将无效：

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

在函数调用中，关键字的参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的某个参数相匹配（例如actor不是parrot函数的有效参数），它们的顺序并不重要。这也包括非可选参数（例如 parrot(voltage=1000) 也是有效的）。任何参数都不可以多次赋值。下面的示例由于这种限制将失败：

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

当最后一个形参以**name形式出现时，它接收一个字典(见映射类型 — 字典)，该字典包含了所有未出现在形式参数列表中的关键字参数。它还可能与*name*形式的参数（在下一小节中所述）组合使用，name接收一个包含所有没有出现在形式参数列表中的位置参数元组。(*name必须出现在name之前。)例如，如果我们定义这样的函数：

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

它可以这样调用：

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

当然它会打印：

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

注意在打印关键字参数之前，通过对关键字字典 keys() 方法的结果进行排序，生成了关键字参数名的列表；如果不这样做，打印出来的参数的顺序是未定义的。

4.7.3.任意参数列表

最后，一个最不常用的场景是让函数可以被可变个数的参数调用。这些参数被放在一个元组（见元组和序列）中。在可变个数的参数之前，可以有零到多个普通的参数。

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

4.7.4. 参数列表的分拆

当传递的参数已经是一个列表或元组时，情况与之前相反，你要分拆这些参数，因为函数调用要求独立的位置参数。例如，内置的range()函数期望单独的*start*和*stop*参数。如果它们不是独立的，函数调用时使用 *-操作符将参数从列表或元组中分拆开来：

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

以同样的方式，可以用**-操作符让字典传递关键字参数：

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demis
```

4.7.5. lambda表达式

可以使用lambda关键字创建小的匿名函数。下面这个函数返回它的两个参数的和：lambdaa,b:a + b。Lambda 函数可以用于任何需要函数对象的地方。在语法上，它们被局限于只能有一个单独的表达式。在语义上，他们只是普通函数定义的语法糖。像嵌套的函数定义，lambda 函数可以从包含范围引用变量：

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

上面的示例使用 lambda 表达式返回一个函数。另一个用途是将一个小函数作为参数传递：

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. 文档字符串

有关文档字符串的内容和格式的约定正不断涌现。

第一行永远应该是对象用途的简短、精确的总述。为了简单起见，不应该明确的陈述对象的名字或类型，因为这些信息可以从别的途径了解到（除非这个名字碰巧就是描述这个函数操作的动词）。这一行应该以大写字母开头，并以句号结尾。

如果在文档字符串中有更多的行，第二行应该是空白，在视觉上把摘要与剩余的描述分离开来。以下各行应该是一段或多段描述对象的调用约定、 其副作用等。

Python 解释器不会从多行的文档字符串中去除缩进，所以必要的时候处理文档字符串的工具应当自己清除缩进。这通过使用以下约定可以达到。第一行 之后 的第一个非空行字符串确定整个文档字符串的缩进的量。（我们不用第一行是因为它通常紧靠着字符串起始的引号，其缩进格式不明晰。）所有行起始的等于缩进量的空格都将被过滤掉。不应该发生缩进较少的行，但如果他们发生，应去除所有其前导空白。留白的长度应当等于扩展制表符的宽度（正常是 8 个空格）。

这里是一个多行文档字符串的示例：

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8. 插曲：编码风格

既然你将要编写更长更复杂的 Python 片段，这是谈一谈 编码风格 的好时机。大多数语言可以编写成（或者更准确地讲，格式化成）不同的风格；其中有一些会比其他风格更具可读性。让你的代码对别人更易读永远是个好想法，养成良好的编码风格对此有很大的帮助。

对于 Python 而言， **PEP 8** 已成为大多数项目遵循的风格指南；它给出了一个高度可读，视觉友好的编码风格。每个 Python 开发者应该阅读一下；这里是为你提取出来的最重要的要点：

- 使用 4 个空格的缩进，不要使用制表符。

4 个空格是小缩进（允许更深的嵌套）和大缩进（易于阅读）之间很好的折衷。制表符会引起混乱，最好弃用。

- 折行以确保其不会超过 79 个字符。

这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件。

- 使用空行分隔函数和类，以及函数内的大块代码。

- 如果可能，注释独占一行。

- 使用 docstrings。

- 运算符周围和逗号后面使用空格，但是括号里侧不加空格： a=f(1,2)+g(3,4)。

- 命名您的类和函数一致 ；惯例是使用驼峰命名法的类和使用 lower_case_with_underscores 的函数和方法 。始终使用self作为方法的第一个参数的名称（关于类和方法的更多信息请参见初识类）。

- 如果希望你的代码在国际化环境中使用，不要使用奇特的编码。简单的 ASCII 在任何情况下永远工作得最好。

脚注

| [1] | 实际上，更好的描述是通过对象的引用调用，因为如果传递的是一个可变的对象，那么调用者可以看到被调用者对它所做的任何改变（插入到一个列表中元素）。 | |-----|-----|

5. 数据结构

本章详细讲述你已经学过的一些知识，并增加一些新内容。

5.1. 深入列表

列表数据类型还有更多的方法。这里是列表对象的所有方法：

list.append(*x*) 添加一个元素到列表的末尾；相当于a[len(a):]=[x]。

list.extend(*L*) 将给定列表中的所有元素附加到另一个列表的末尾；相当于a[len(a):]=L。

list.insert(*i*, *x*) 在给定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，所以 a.insert(0,x) 在列表的最前面插入，a.insert(len(a),x) 相当于 a.append(x)。

list.remove(*x*) 删除列表中第一个值为 *x* 的元素。如果没有这样的元素将会报错。

list.pop([*i*]) 删除列表中给定位置的元素并返回它。如果未指定索引，a.pop() 删除并返回列表中的最后一个元素。（*i* 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示法)。

list.index(*x*) 返回列表中第一个值为 *x* 的元素的索引。如果没有这样的元素将会报错。

list.count(*x*) 返回列表中 *x* 出现的次数。

list.sort(*cmp=None*, *key=None*, *reverse=False*) 原地排序列表中的元素（参数可以用来自定义排序方法，参考sorted()的更详细的解释）。

list.reverse() 原地反转列表中的元素。

使用了列表大多数方法的例子：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

你可能已经注意到像insert, remove 或者 sort之类的方法只修改列表而没有返回值打印出来 -- 它们其实返回了默认值None。这是 Python 中所有可变数据结构的设计原则。

5.1.1. 用列表作为栈

列表方法使得将List当作栈非常容易，最先进入的元素最后一个取出（后进先出）。使用append()将元素添加到栈顶。使用不带索引的pop()从栈顶取出元素。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. 用列表作为队列

也可以将列表当作队列使用，此时最先进入的元素第一个取出（先进先出）；但是列表用作此目的效率不高。在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素必须向后移一位)。

如果要实现一个队列，可以使用collections.deque，它设计的目的就是在两端都能够快速添加和弹出元素。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. 函数式编程工具

有三个内置函数与列表一起使用时非常有用：filter()、map()和reduce()。

filter(function,sequence)返回的序列由function(item)结果为真的元素组成。如果*sequence*是一个字符串或元组，结果将是相同的类型；否则，结果将始终是一个列表。例如，若要计算一个不能被2和3整除的序列：

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map(function,sequence) 为序列中的每一个元素调用 function(item) 函数并返回结果的列表。例如，计算列表中所有元素的立方值：

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

可以传入多个序列；此时，传入的函数也必须要有和序列数目相同的参数，执行时会依次用各序列上对应的元素来调用函数（如果某个序列比另外一个短，就用 None 代替）。例如：

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

reduce(function,sequence) 只返回一个值，它首先以序列的前两个元素调用函数 *function*，然后再以返回的结果和下一个元素继续调用，依此执行下去。例如，若要计算数字 1 到 10 的总和：

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

如果序列中只有一个元素，将返回这个元素的值；如果序列为空，则引发异常。

可以传入第三个参数作为初始值。在这种情况下，如果序列为空则返回起始值，否则会首先以初始值和序列的第一个元素调用function，然后是返回值和下一个元素，依此执行下去。例如，

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

不要使用示例中定义的sum()：由于计算数字的总和是一个如此常见的需求，Python提供了内置的函数sum(sequence)，其工作原理和示例几乎一样。

5.1.4. 列表推导式

列表推导式提供了一个生成列表的简洁方法。应用程序通常会从一个序列的每个元素的操作结果生成新的列表，或者生成满足特定条件的元素的子序列。

例如，假设我们想要创建一个列表 squares：

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

我们可以用下面的方式得到同样的结果：

```
squares = [x**2 for x in range(10)]
```

这也相当于squares =map(lambdax:x**2,range(10))，但是更简洁和易读。

列表推导式由括号括起来，括号里面包含一个表达式，表达式后面跟着一个for语句，后面还可以接零个或更多的for或if语句。结果是一个新的列表，由表达式依据其后面的for和if子句上下文计算而来的结果构成。例如，下面的 listcomp 组合两个列表中不相等的元素：

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

它等效于：

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意在两个代码段中for 和 if 语句的顺序是相同的。

如果表达式是一个元组（例如前面示例中的 (x，y)），它必须带圆括号。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表推导式可以包含复杂的表达式和嵌套的函数：

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4.1. 嵌套的列表推导式

列表推导式中的第一个表达式可以是任何表达式，包括另外一个列表推导式。

考虑下面由三个长度为 4 的列表组成的 3x4 矩阵：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

下面的列表推导式将转置行和列：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

正如我们在上一节中看到的，嵌套的 listcomp 在跟随它之后的for 字句中计算，所以此例等同于：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以此下去，还等同于：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在实际中，与复杂的控制流比起来，你应该更喜欢内置的函数。针对这种场景，使用 zip() 函数会更好：

```
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

关于本行中使用的星号的说明，请参阅参数列表的分拆。

5.2. del语句

有个方法可以从列表中根据索引而不是值来删除一个元素：del语句。这不同于有返回值的 pop()方法。del语句还可以用于从列表中删除切片或清除整个列表（之前我们是将空列表赋值给切片）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del也可以用于删除整个变量：

```
>>> del a
```

此后再引用名称 a 将会报错（直到有另一个值被赋给它）。稍后我们将看到del的其它用途。

5.3. 元组和序列

我们已经看到列表和字符串具有很多共同的属性，如索引和切片操作。它们是序列数据类型的两个例子（参见*序列类型 — str, unicode, list, tuple, bytearray, buffer, xrange*)。因为 Python 是一个正在不断进化的语言，其他的序列类型也可能被添加进来。还有另一种标准序列数据类型：元组。

元组由逗号分割的若干值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构；在输入时可以有也可以没有括号，不过括号经常都是必须的（如果元组是一个更大的表达式的一部分）。不能给元组中单独的一个元素赋值，不过可以创建包含可变对象（例如列表）的元组。

虽然元组看起来类似于列表，它们经常用于不同的场景和不同的目的。元组是不可变的，通常包含不同种类的元素并通过分拆（参阅本节后面的内容）或索引访问（如果是namedtuples，甚至可以通过属性）。列表是可变的，它们的元素通常是相同的类型并通过迭代访问。

一个特殊的情况是构造包含0个或1个元素的元组：为了实现这种情况，语法上有一些奇怪。空元组由一对空括号创建；只有一个元素的元组由值后面跟随一个逗号创建 (在括号中放入单独一个值还不够）。丑陋，但是有效。例如：

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句t=12345,54321,'hello!' 是一个元组封装的例子： 值12345,54321 和 'hello!' 被一起放入一个元组。其逆操作也是可以的：

```
>>> x, y, z = t
```

这被称为 序列分拆 再恰当不过了，且可以用于右边的任何序列。序列分拆要求左侧变量的数目和序列中元素的数目相同。注意多重赋值只是同时进行元组封装和序列分拆。

5.4. 集合

Python还包含一个数据类型用于集合。集合中的元素没有顺序且不会重复。集合的基本用途有成员测试和消除重复的条目。集合对象还支持并集、交集、差和对称差等数学运算。

花括号或set()函数可以用于创建集合。注意：若要创建一个空的集合你必须使用set()，不能用{}；后者将创建一个空的字典，一个我们在下一节中要讨论的数据结构。

这里是一个简短的演示：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # letters in both a and b
set(['a', 'c'])
>>> a ^ b                              # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

和列表推导式类似，集合也支持推导式：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

5.5. 字典

Python内建的另一种有用的数据类型是字典（见映射类型 — 字典)。在其它语言中字典有时被称为“associative memories” 或者 “associative arrays”。与序列不同，序列由数字做索引，字典由 *key* 做索引，key可以是任意不可变类型；字符串和数字永远可以拿来做key。如果元组只包含字符串、 数字或元组，此元组可以用作key；如果元组直接或间接地包含任何可变对象，那么它不能用作键。不能用列表作为键，因为列表可以用索引、切片或者append()和extend()方法原地修改。

理解字典的最佳方式是把它看做无序的键:值 对集合，要求是键必须是唯一的（在同一个字典内）。一对花括号将创建一个空的字典：{}。花括号中由逗号分隔的键:值对将成为字典的初始值；打印字典时也是按照这种方式输出。

字典的主要操作是依据键来存取值。也可以通过del删除键:值对。如果用一个已经存在的键存储值，以前为该关键字分配的值就会被遗忘。用一个不存在的键读取值会导致错误。

字典对象的keys()方法返回字典中所有键组成的列表，列表的顺序是随机的（如果你想要排序，只需在它上面调用sorted()函数）。要检查某个键是否在字典中，可以使用in关键字。

下面是一个使用字典的小示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

dict()构造函数直接从键-值对序列创建字典：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导式式可以用于从任意键和值表达式创建字典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果键都是简单的字符串，有时通过关键字参数指定 键-值 对更为方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6. 遍历的技巧

遍历一个序列时，使用enumerate()函数可以同时得到索引和对应的值。

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

同时遍历两个或更多的序列，使用zip()函数可以成对读取元素。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先正向生成这个序列，然后调用 reversed() 函数。

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

循环一个序列按排序顺序，请使用sorted()函数，返回一个新的排序的列表，同时保留源不变。

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

遍历字典时，使用iteritems()方法可以同时得到键和对应的值。

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

若要在循环内部修改正在遍历的序列（例如复制某些元素），建议您首先制作副本。在序列上循环不会隐式地创建副本。切片表示法使这尤其方便：

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.7. 深入条件控制

while 和 if 语句中使用的条件可以包含任意的操作，而不仅仅是比较。

比较操作符 in 和 notin 检查一个值是否在一个序列中出现（不出现）。is 和 isnot 运算符比较两个对象是否为相同的对象；这只和列表这样的可变对象有关。所有比较运算符都具有相同的优先级，低于所有数值运算符。

比较可以级联。例如，a<b==c测试a是否小于b并且b是否等于c。

可以使用布尔运算符 and 和 or 组合，比较的结果（或任何其他的布尔表达式）可以用 not 取反。这些操作符的优先级又低于比较操作符；它们之间，not 优先级最高，or 优先级最低，所以 AandnotBorC 等效于 (Aand(notB))orC。与往常一样，可以使用括号来表示所需的组合。

布尔运算符and 和 or 是所谓的 短路 运算符：依参数从左向右求值，结果一旦确定就停止。例如，如果A 和 C 都为真，但B是假， AandBandC 将不计算表达式 C。用作一个普通值而非逻辑值时，短路操作符的返回值通常是最后一个计算的。

可以把比较或其它逻辑表达式的返回值赋给一个变量。例如，

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

注意在Python中，与C不同，表达式的内部不能出现赋值。C程序员可能会抱怨这一点，但它避免了C程序中常见的一类问题：在表达式中输入=而真正的意图是==。

5.8. 序列和其它类型的比较

序列对象可以与具有相同序列类型的其他对象相比较。比较按照 字典序 进行： 首先比较两个序列的首元素，如果不同，就决定了比较的结果；如果相同，就比较后面两个元素，依此类推，直到其中一个序列穷举完。如果要比较的两个元素本身就是同一类型的序列，就按字典序递归比较。如果两个序列的所有元素都相等，就为序认列相等。如果一个序列是另一个序列的初始子序列，较短的序列就小于另一个。字符串的字典序按照单字符的 ASCII 顺序。下面是同类型序列之间比较的一些例子：

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0
```

6. 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。因此，如果你想要编写一些更大的程序，最好使用文本编辑器先编写好，然后运行这个文件。这就是所谓的创建 脚本。随着你的程序变得越来越长，你可能想要将它分成几个文件，这样更易于维护。你还可能想在几个程序中使用你已经编写好的函数，而不用把函数拷贝到每个程序中。

为了支持这个功能，Python 有种方法可以把你定义的内容放到一个文件中，然后在脚本或者交互方式中使用。这种文件称为模块；模块中的定义可以 导入 到其它模块或 主模块 中。

模块是包含 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。在模块里面，模块的名字（是一个字符串）可以由全局变量 **name** 的值得到。例如，用你喜欢的文本编辑器在当前目录下创建一个名为 fibo.py 的文件，内容如下：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用下面的命令导入这个模块：

```
>>> import fibo
```

这不会直接把 fibo 中定义的函数的名字导入当前的符号表中；它只会把模块名字 fibo 导入其中。你可以通过模块名访问这些函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算频繁使用一个函数，可以将它赋给一个本地的变量：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. 深入模块

模块可以包含可执行语句以及函数的定义。这些语句通常用于初始化模块。它们只在 第一次 导入时执行。[1]（如果文件以脚本的方式执行，它们也会运行。）

每个模块都有自己的私有符号表，模块内定义的所有函数用其作为全局符号表。因此，模块的作者可以在模块里使用全局变量，而不用担心与某个用户的全局变量有冲突。另一方面，如果你知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量，modname.itemname。

模块中可以导入其它模块。习惯上将所有的 import 语句放在模块（或者脚本）的开始，但这不是强制性的。被导入的模块的名字放在导入模块的全局符号表中。

import 语句的一个变体直接从被导入的模块中导入名字到导入模块的符号表中。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这不会把模块名导入到本地的符号表中（所以在本例中，fibo 将没有定义）。

还有种方式可以导入模块中定义的所有名字：

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式不会导入以下划线 (_) 开头的名称。

注意一般情况下不赞成从一个模块或包中导入 *，因为这通常会导致代码很难读。不过，在交互式会话中这样用是可以的，它可以让你少敲一些代码。

注意

出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，你必需重新启动解释器 —— 或者，如果你就是想交互式的测试这么一个模块，可以使用reload()，例如reload(modulename)。

6.1.1. 执行模块

当你用下面的方式运行一个 Python 模块

```
python fibo.py <arguments>
```

模块中的代码将会被执行，就像导入它一样，不过此时 **name** 被设置为 **"main"** 。也就是说，如果你在模块后加入如下代码:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

就可以让此文件既可以作为可执行的脚本，也可以当作可以导入的模块，因为解析命令行的那部分代码只有在模块作为 “main” 文件执行时才被调用：

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

如果模块是被导入的，将不会运行这段代码：

```
>>> import fibo
>>>
```

这种方法通常用来为模块提供一个方便的用户接口，或者用来测试（例如直接运行脚本会执行一组测试用例）。

6.1.2. 模块搜索路径

当导入一个名为 spam 的模块时，解释器首先搜索具有该名称的内置模块。如果没有找到，它会接着到 sys.path 变量给出的目录中查找名为 spam.py 的文件。sys.path 变量的初始值来自这些位置：

- 脚本所在的目录（或当前目录）。
- PYTHONPATH （一个包含目录名的列表，与 shell 变量PATH 的语法相同）。
- 与安装相关的默认值。

初始化后，Python 程序可以修改sys.path。脚本所在的目录被放置在搜索路径的最开始，也就是在标准库的路径之前。这意味着将会加载当前目录中的脚本，库目录中具有相同名称的模块不会被加载。除非你是有意想替换标准库，否则这应该被当成是一个错误。更多信息请参阅 标准模块 小节。

6.1.3. "编译过的" Python 文件

对于使用了大量标准模块的简短程序，有一个提高启动速度的重要方法，如果在spam.py所在的目录下存在一个名为spam.pyc的文件，它会被视为spam模块的已“编译”版本。生成spam.pyc文件时，spam.py文件的修改时间会被记录在spam.pyc文件中，如果时间不匹配，.pyc文件将被忽略。

通常情况下，您不需要做任何事情来创建spam.pyc文件。每当spam.py编译成功，会尝试向spam.pyc写入编译的版本。尝试失败也不会出现错误；如果出于任何原因文件写入不完全，生成的 spam.pyc 文件将被当作是无效的，并且在以后会被忽略。spam.pyc文件的内容与平台无关，因此Python模块的目录可以在不同体系结构的机器间共享。

部分高级技巧：

- 当以-O标志调用Python解释器时，将生成优化的代码并保存在.pyo文件中。目前的优化不会帮助太多；它只是删除assert语句。当使用-O时，将优化所有的字节码；将忽略.pyc文件并将.py文件编译成优化的字节码。
- 向 Python 解释器传递两个-O标志 （-OO） 会导致字节码编译器执行优化，极少数情况下，这可能导致程序故障。目前只会从字节码中删除**doc**字符串，使.pyo文件更加紧凑。因为某些程序可能会依赖于具有这些可用，您应只使用此选项，在你知道你在做什么时。
- 程序不能运行得更快，不管它是从哪种文件（.py ； .pyc或.pyo）中读取。唯一加速的是.pyc或.pyo文件的加载速度。
- 当在命令行使用脚本名称来运行脚本时，脚本的字节码是不会被写入相应的.pyc或.pyo文件中的。因此，如果将大部分与启动无关的代码移到一个模块中而以导入模块的方式在启动脚本中导入这个模块就可节省一些启动时间。也可以直接在命令行上运行.pyc或.pyo文件。
- 可以没有相同的模块文件spam.py时，使用文件spam.pyc （或spam.pyo时使用-O ）。这可用于发布不太容易进行反向工程的Python 代码库。
- 模块compileall可以为一个目录中的所有模块创建.pyc文件 （或.pyo文件时使用-O ）。

6.2. 标准模块

Python 带有一个标准模块库，并发布有单独的文档叫Python 库参考手册（以下简称"库参考手册"）。有些模块被直接构建在解析器里；这些操作虽然不是语言核心的部分，但是依然被内建进来，一方面是效率的原因，另一方面是为了提供访问操作系统原语如系统调用的功能。这些模块是可配置的，也取决于底层的平台。例如，winreg模块只在Windows系统上提供。有一个特别的模块值得注意：sys，它内置在每一个Python解析器中。变量sys.ps1和sys.ps2定义了主提示符和辅助提示符使用的字符串：

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

只有在交互式模式中，这两个变量才有定义。

变量 sys.path 是一个字符串列表，它决定了解释器搜索模块的路径。它初始的默认路径来自于环境变量 PYTHONPATH，如果 PYTHONPATH 未设置则来自于内置的默认值。你可以使用标准的列表操作修改它：

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. dir()函数

内置函数 dir() 用来找出模块中定义了哪些名字。它返回一个排好序的字符串列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 'version', 'version_info', 'warnoptions']
```

如果不带参数， dir() 列出当前已定义的名称：

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

注意它列出了所有类型的名称：变量、模块、函数等。

dir()不会列出内置的函数和变量的名称。如果你想列出这些内容，它们定义在标准模块 **builtin** 中：

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4. 包

包是一种管理 Python 模块命名空间的方式，采用“点分模块名称”。例如，模块名 A.B 表示包 A 中一个名为 B 的子模块。就像模块的使用让不同模块的作者不用担心相互间的全局变量名称一样，点分模块的使用让包含多个模块的包（例如 Numpy 和 Python Imaging Library）的作者也不用担心相互之间的模块重名。

假设你想要设计一系列模块（或一个“包”）来统一处理声音文件和声音数据。现存很多种不同的声音文件格式 （通常由它们的扩展名来识别，例如： .wav, .aiff, .au），因此你可能需要创建和维护不断增长的模块集合来支持各种文件格式之间的转换。你可能还想针对音频数据做很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要编写一组永远写不完的模块来处理这些操作。你的包可能会是这个结构（用分层的文件系统表示）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

导入这个包时，Python 搜索 sys.path 中的目录以寻找这个包的子目录。

为了让 Python 将目录当做包，目录下必须包含 **init**.py 文件；这样做是为了防止一个具有常见名字（例如 string）的目录无意中隐藏目录搜索路径中正确的模块。最简单的情况下，**init**.py 可以只是一个空的文件，但它也可以为包执行初始化代码或设置**all**变量（稍后会介绍）。

用户可以从包中导入单独的模块，例如：

```
import sound.effects.echo
```

这样就加载了子模块sound.effects.echo。它必须使用其完整名称来引用。

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

导入子模块的另一方法是：

```
from sound.effects import echo
```

这同样也加载了子模块echo，但使它可以不用包前缀访问，因此它可以按如下方式使用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有另一种变化方式是直接导入所需的函数或变量：

```
from sound.effects.echo import echofilter
```

这再次加载了子模块echo，但这种方式使函数echofilter() 可以直接访问：

```
echofilter(input, output, delay=0.7, atten=4)
```

注意使用frompackageimportitem时，item 可以是包的子模块（或子包），也可以是包中定义的一些其它的名称，比如函数、 类或者变量。import语句首先测试 item 在包中是否有定义；如果没有，它假定它是一个模块，并尝试加载它。如果未能找到，则引发ImportError异常。

相反，使用类似 importitem.subitem.subsubitem 这样的语法时，除了最后一项其它每项必须是一个包；最后一项可以是一个模块或一个包，但不能是在前一个项目中定义的类、函数或变量。

6.4.1. 从包中导入 *

那么现在如果用户写成 fromsound.effectsimport* 会发生什么？理想情况下，他应该是希望到文件系统中寻找包里面有哪些子模块，并把它们全部导入进来。这可能需要很长时间，而且导入子模块可能会产生想不到的副作用，这些作用本应该只有当子模块是显式导入时才会发生。

唯一的解决办法是包的作者为包提供显式的索引。import 语句使用以下约定： 如果包中的 **init**.py 代码定义了一个名为**all**的列表，那么在遇到 frompackageimport语句的时候，应该把这个列表中的所有模块名字导入。当包有新版本包发布时，就需要包的作者更新这个列表了。如果包的作者认为不可以用 *import* 方式导入它们的包，也可以决定不支持它。例如，文件sound/effects/**init**.py可以包含下面的代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这意味着 fromsound.effectsimport* 将导入sound 包的三个子模块。

如果 **all** 没有定义，fromsound.effectsimport 语句 不* 会从 sound.effects 包中导入所有的子模块到当前命名空间；它只保证 sound.effects 包已经被导入（可能会运行 **init**.py 中的任何初始化代码），然后导入包中定义的任何名称。这包括由 **init**.py 定义的任何名称（以及它显式加载的子模块）。还包括这个包中已经由前面的import 语句显式加载的子模块。请考虑此代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

在这个例子中，执行 from...import 语句时，echo 和 surround 模块被导入到当前命名空间是因为它们在sound.effects中有定义。（定义了 **all**时也会同样工作。）

虽然某些模块设计成使用 import* 时只导出符合特定模式的名称，在产品代码中使用这种写法仍然是不好的做法。

记住，使用 fromPackageimportspecific_submodule 一点没错 ！事实上，这是推荐的写法，除非导入的模块需要使用其它包中的同名子模块。

6.4.2. 包内引用

子模块通常需要相互引用。例如，surround 模块可能会使用 echo 模块。事实上，这种引用是如此常见以致 import 语句会在标准模块搜索路径之前首先在所在的包中查找。因此，surround 模块可以简单地使用importecho 或 fromechoimportechofilter。如果在当前包（当前模块属于其子模块的包）中未找到要导入的模块，import 语句会查找具有给定名称的顶级模块。

如果一个包是子包（比如例子中的 sound 包），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块 sound.filters.vocoder 需要使用sound.effects 包中的 echo 模块，它可以使用 fromsound.effectsimportecho。

Python 2.5 开始，除了上面描述的隐式相对导入，你可以使用frommoduleimportname 的导入形式编写显式相对导入。这些显式的相对导入使用前导的点号表示相对导入的是从当前包还是上级的包。以 surround 模块为例，你可以使用：

```
from . import echo
from .. import formats
from ..filters import equalizer
```

注意，显式和隐式相对导入都基于当前模块的名称。因为主模块的名字总是 "**main**"，Python 应用程序的主模块应该总是用绝对导入。

6.4.3. 包含多个目录的包

包还支持一个特殊的属性， **path**。在文件运行之前，该变量被初始化为一个包含 **init**.py 所在目录的列表。这个变量可以修改；这样做会影响未来包中包含的模块和子包的搜索。

虽然通常不需要此功能，它可以用于扩展包中的模块的集合。

脚注

| [1] | 实际上，函数的定义也是‘执行过’的‘语句’；模块级别的函数定义的执行将函数名放入该模块的全局符号表中。 | |-----|-----|

7. 输入和输出

展现程序的输出有多种方法；可以打印成人类可读的形式，也可以写入到文件以便后面使用。本章将讨论其中的几种方法。

7.1. 格式化输出

到目前为止我们遇到过两种输出值的方法：表达式语句和print语句。（第三个方式是使用文件对象的write()方法；标准输出文件可以引用 sys.stdout。详细内容参见库参考手册。）

通常你会希望更好地控制输出的格式而不是简单地打印用空格分隔的值。有两种方法来设置输出格式；第一种方式是自己做所有的字符串处理；使用字符串切片和连接操作，你可以创建任何你能想象到的布局。字符串类型有一些方法，用于执行将字符串填充到指定列宽度的有用操作；这些稍后将讨论。第二种方法是使用str.format()方法。

string模块包含一个Template类，提供另外一种向字符串代入值的方法。

当然还有一个问题：如何将值转换为字符串？幸运的是，Python 有方法将任何值转换为字符串：将它传递给repr()或str()函数。

str()函数的用意在于返回人类可读的表现形式，而repr()的用意在于生成解释器可读的表现形式（如果没有等价的语法将会引发SyntaxError异常）。对于对人类并没有特别的表示形式的对象， str()和repr()将返回相同的值。许多值，例如数字或者列表和字典这样的结构，使用这两个函数中的任意一个都具有相同的表示形式。特别地，字符串和浮点数有两种不同的表示形式。

一些例子：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

这里用两种方法输出平方和立方表：

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

（注意在第一个示例中，每列之间的一个空格由print自动添加：它总会在它的参数之间添加空格。）

上面的例子演示了字符串对象的str.rjust()方法，它通过在左侧填充空格使字符串在给定宽度的列右对齐。类似的方法还有str.ljust()和str.center()。这些方法不会输出任何内容，它们只返回新的字符串。如果输入的字符串太长，它们不会截断字符串，而是保持原样返回；这会使列的格式变得混乱，但是通常好于另外一种选择，那可能是一个错误的值。（如果你真的想要截断，可以加上一个切片操作，例如x.ljust(n)[:n]。）

另外一种方法 str.zfill()，它向数值字符串左侧填充零。该函数可以正确识别正负号：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format()方法的基本用法如下所示：

```
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

花括号及其中的字符（称为格式字段）将被替换为传递给str.format()方法的对象。括号中的数字指传递给str.format()方法的对象的位置。

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

如果str.format()方法使用关键字参数，那么将通过参数名引用它们的值。

```
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

位置参数和关键字参数可以随意组合：

```
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                    other='Georg')
The story of Bill, Manfred, and Georg.
```

'!s'（适用str()）和'!r'（适用repr()）可以用于值被格式化之前对值进行转换。

```
>>> import math
>>> print 'The value of PI is approximately {}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

字段名后允许可选的':'和格式指令。这允许更好地控制如何设置值的格式。下面的例子将 Pi 转为三位精度。

```
>>> import math
>>> print 'The value of PI is approximately {0:.3f}.'.format(math.pi)
The value of PI is approximately 3.142.
```

':'后面紧跟一个整数可以限定该字段的最小宽度。这在美化表格时很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

如果你有一个实在是很长的格式字符串但又不想分开写，要是可以按照名字而不是位置引用变量就好了。有个简单的方法，可以传入一个字典，然后使用'[]'访问。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...        'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这也可以用'**'符号将这个字典以关键字参数的方式传入。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这种方式与内置函数vars()组合起来将更加有用，该函数返回一个包含所有局部变量的字典。

关于str.format()完整的描述，请参见格式字符串语法。

7.1.1. 旧式的字符串格式

%运算符也可以用于字符串格式化。它将左边类似sprintf()-风格的参数应用到右边的参数，然后返回这种格式化操作生成的字符串。例如：

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

字符串格式化操作 一节中，可以找到更多的信息。

7.2. 读写文件

open()返回一个文件对象，最常见的用法带有两个参数：open(filename,mode)。

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

第一个参数是一个含有文件名的字符串。第二个参数也是一个字符串，含有描述如何使用该文件的几个字符。*mode*为'r'时表示只是读取文件；w 表示只是写入文件（已经存在的同名文件将被删掉）；'a'表示打开文件进行追加，写入到文件中的任何数据将自动添加到末尾。'r+'表示打开文件进行读取和写入。*mode* 参数是可选的，默认为'r'。

在 Windows 平台上，模式后面追加 'b'表示以二进制方式打开文件，所以也有像'rb'、'wb'和'r+b'这样的模式。Python 在 Windows 平台上区分文本文件和二进制文件；读取或写入文本文件中时，行尾字符会被自动地稍加改变。这种修改对 ASCII 文本文件没有问题，但会损坏JPEG或EXE这样的二进制文件中的数据。在读写这些文件时一定要记得以二进制模式打开。在 Unix 平台上，在模式后面附加一个'b'不会有损坏，因此你可以用它来读写任何平台上的所有二进制文件。

7.2.1. 文件对象的方法

本节中的示例将假设文件对象f已经创建。

要读取文件内容，可以调用f.read(size)，该方法读取若干数量的数据并以字符串形式返回其内容。*size* 是可选的数值参数。当 *size* 被省略或者为负数时，将会读取并返回整个文件；如果文件大小是你机器内存的两倍时，就是你的问题了。否则，至多读取和返回 *size* 大小的字节数据。如果到了文件末尾，f.read() 会返回一个空字符串("")。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline()从文件读取一行数据；字符串结尾会带有一个换行符 (\n)，只有当文件最后一行没有以换行符结尾时才会省略。这样返回值就不会有混淆，如果 f.readline()返回一个空字符串，那就表示已经达到文件的末尾，而如果返回一个只包含一个换行符的字符串'\n'，则表示遇到一个空行。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

你可以循环遍历文件对象来读取文件中的每一行。这是既省内存又非常快的简单代码：

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

如果你想把文件中的所有行读到一个列表中，你也可以使用list(f)或f.readlines()。

f.write(string)将 *string* 的内容写入文件中并返回None。

```
>>> f.write('This is a test\n')
```

如果想写入字符串以外的数据，需要先将它转换为一个字符串：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

f.tell()返回一个整数，代表文件对象在文件中的指针位置，该数值计量了自文件开头到指针处的比特数。若要更改该文件对象的位置，可以使用f.seek(offset,from_what)。新的位置由参考点加上 *offset* 计算得来，参考点的选择则来自于 *from_what* 参数。*from_what* 值为 0 表示以文件的开始为参考点，1 表示以当前的文件位置为参考点，2 表示以文件的结尾为参考点。*from_what* 可以省略，默认值为 0，表示以文件的开始作为参考点。

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

使用完一个文件后，调用f.close()可以关闭它并释放其占用的所有系统资源。调用f.close()后，再尝试使用该文件对象将失败。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

处理文件对象时使用with关键字是很好的做法。这样做的好处在于文件用完后会自动关闭，即使过程中发生异常也没关系。它还比编写一个等同的try-finally语句要短很多：

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有一些不太常用的方法，例如isatty()和truncate()；有关文件对象的完整指南，请参阅 Python 库参考手册。

7.2.2. 使用json存储结构化数据

从文件中读写字符串很容易。数值就要多费点儿周折，因为read ()方法只会返回字符串，应将其传入int()这样的函数，就可以将'123'这样的字符串转换为对应的数值 123。当你想要保存更为复杂的数据类型，例如嵌套的列表和字典，手工解析和序列化它们将变得更复杂。

好在用户不是非得自己编写和调试保存复杂数据类型的代码，Python 允许你使用常用的数据交换格式JSON（JavaScript Object Notation）。标准模块json可以接受 Python 数据结构，并将它们转换为字符串表示形式；此过程称为序列化。从字符串表示形式重新构建数据结构称为反序列化。序列化和反序列化的过程中，表示该对象的字符串可以存储在文件或数据中，也可以通过网络连接传送给远程的机器。

注意

JSON 格式经常用于现代应用程序中进行数据交换。许多程序员都已经熟悉它了，使它成为相互协作的一个不错的选择。

如果你有一个对象x，你可以用简单的一行代码查看其 JSON 字符串表示形式：

```
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

dumps()函数的另外一个变体dump()，直接将对象序列化到一个文件。所以如果f是为写入而打开的一个文件对象，我们可以这样做：

```
json.dump(x, f)
```

为了重新解码对象，如果f是为读取而打开的文件对象 ：

```
x = json.load(f)
```

这种简单的序列化技术可以处理列表和字典，但序列化任意类实例为 JSON 需要一点额外的努力。Json模块的手册对此有详细的解释。

另请参阅

pickle - pickle模块

与*JSON* 不同，*pickle* 是一个协议，它允许任意复杂的 Python 对象的序列化。因此，它只能用于 Python 而不能用来与其他语言编写的应用程序进行通信。默认情况下它也是不安全的：如果数据由熟练的攻击者精心设计， 反序列化来自一个不受信任源的 pickle 数据可以执行任意代码。

8. 错误和异常

直到现在，我们还没有更多的提及错误信息，但是如果你真的尝试了前面的例子，也许你已经见到过一些。Python（至少）有两种错误很容易区分：语法错误 和异常。

8.1. 语法错误

语法错误，或者称之为解析错误，可能是你在学习 Python 过程中最烦的一种：

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的'箭头'。错误是由箭头前面 的标记引起的（至少检测到是这样的）： 在这个例子中，检测到关键字print有问题，因为在它之前缺少一个冒号（':'）。文件名和行号会一并输出，所以如果运行的是一个脚本你就知道去哪里检查错误了。

8.2. 异常

即使一条语句或表达式在语法上是正确的，在运行它的时候，也有可能发生错误。在执行期间检测到的错误被称为异常 并且程序不会无条件地崩溃：你很快就会知道如何在 Python 程序中处理它们。然而大多数异常都不会被程序处理，导致产生类似下面的错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

最后一行的错误消息指示发生了什么事。异常有不同的类型，其类型会作为消息的一部分打印出来：在这个例子中的类型有ZeroDivisionError、 NameError和TypeError。打印出来的异常类型的字符串就是内置的异常的名称。这对于所有内置的异常是正确的，但是对于用户自

定义的异常就不一定了（尽管这是非常有用的惯例）。标准异常的名称都是内置的标识符（不是保留的关键字）。

这一行最后一部分给出了异常的详细信息和引起异常的原因。

错误信息的前面部分以堆栈回溯的形式显示了异常发生的上下文。通常调用栈里会包含源代码的行信息，但是来自标准输入的源码不会显示行信息。

内置的异常 列出了内置的异常以及它们的含义。

8.3. 处理异常

可以通过编程来选择处理部分异常。看一下下面的例子，它会一直要求用户输入直到输入一个合法的整数为止，但允许用户中断这个程序（使用Control-C或系统支持的任何方法）；注意用户产生的中断引发的是 KeyboardInterrupt 异常。

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

Try语句按以下方式工作。

- 首先，执行*try* 子句（try和except关键字之间的语句）。
- 如果未发生任何异常，忽略*except* 子句且try语句执行完毕。
- 如果在 try 子句执行过程中发生异常，跳过该子句的其余部分。如果异常的类型与except关键字后面的异常名匹配, 则执行 except 子句，然后继续执行try语句之后的代码。
- 如果异常的类型与 except 关键字后面的异常名不匹配，它将被传递给上层的try语句；如果没有找到处理这个异常的代码，它就成为一个未处理异常，程序会终止运行并显示一条如上所示的信息。

Try语句可能有多个子句，以指定不同的异常处理程序。不过至多只有一个处理程序将被执行。处理程序只处理发生在相应 try 子句中的异常，不会处理同一个try字句的其他处理程序中发生的异常。一个 except 子句可以用带括号的元组列出多个异常的名字，例如：

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

注意，此元组周围的括号是必需的，因为exceptValueError,e:是旧式的写法，在现代 Python 中通常写成 exceptValueErrorase: （如下所述）。为了保持向后兼容性，旧式语法仍然是支持的。这意味着exceptRuntimeError,TypeError不等同于except(RuntimeError,TypeError): 而

等同于exceptRuntimeErrorasTypeError: , 这应该不是你想要的。

最后一个 except 子句可以省略异常名称，以当作通配符使用。使用这种方式要特别小心，因为它会隐藏一个真实的程序错误！它还可以用来打印一条错误消息，然后重新引发异常 （让调用者也去处理这个异常）：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

try...except语句有一个可选的*else* 子句，其出现时，必须放在所有 except 子句的后面。如果需要在 try 语句没有抛出异常时执行一些代码，可以使用这个子句。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

使用else子句比把额外的代码放在try子句中要好，因为它可以避免意外捕获不是由try ... except语句保护的代码所引发的异常。

当异常发生时，它可能带有相关数据，也称为异常的参数。参数的有无和类型取决于异常的类型。

except 子句可以在异常名（或元组）之后指定一个变量。这个变量将绑定于一个异常实例，同时异常的参数将存放在实例的args中。为方便起见，异常实例定义了**str()**，因此异常的参数可以直接打印而不必引用.args。

也可以在引发异常之前先实例化一个异常，然后向它添加任何想要的属性。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to be printed directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于未处理的异常，如果它含有参数，那么参数会作为异常信息的最后一部分打印出来。

异常处理程序不仅处理直接发生在 try 子句中的异常，而且还处理 try 子句中调用的函数（甚至间接调用的函数）引发的异常。例如：

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

8.4. 引发异常

raise语句允许程序员强行引发一个指定的异常。例如：

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise的唯一参数指示要引发的异常。它必须是一个异常实例或异常类（从Exception派生的类）。

如果你确定需要引发异常，但不打算处理它，一个简单形式的raise语句允许你重新引发异常：

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5. 用户定义的异常

程序可以通过创建新的异常类来命名自己的异常（Python 类的更多内容请参见类）。异常通常应该继承Exception类，直接继承或者间接继承都可以。例如：

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在此示例中，Exception默认的**init()**被覆盖了。新的行为简单地创建了*value* 属性。这将替换默认的创建*args* 属性的行为。

异常类可以像其他类一样做任何事情，但是通常都会比较简单，只提供一些属性以允许异常处理程序获取错误相关的信息。创建一个能够引发几种不同错误的模块时，一个通常的做法是为该模块定义的异常创建一个基类，然后基于这个基类为不同的错误情况创建特定的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

大多数异常的名字都以"Error"结尾，类似于标准异常的命名。

很多标准模块中都定义了自己的异常来报告在它们所定义的函数中可能发生的错误。类 这一章给出了类的详细信息。

8.6. 定义清理操作

Try语句有另一个可选的子句，目的在于定义必须在所有情况下执行的清理操作。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```

不管有没有发生异常，在离开try语句之前总是会执行*finally* 子句。当try子句中发生了一个异常，并且没有except字句处理（或者异常发生在try或else子句中），在执行完finally子句后将重新引发这个异常。try语句由于break、contine或return语句离开时，同样会执行finally子句。以下是一个更复杂些的例子 (同时有except和finally字句的try语句的工作方式与 Python 2.5 一样）：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

正如您所看到的，在任何情况下都会执行finally子句。由两个字符串相除引发的 TypeError异常没有被except子句处理，因此在执行finally子句后被重新引发。

在真实的应用程序中， finally子句用于释放外部资源（例如文件或网络连接），不管资源的使用是否成功。

8.7. 清理操作的预定义

有些对象定义了在不需要该对象时的标准清理操作，无论该对象的使用是成功还是失败。看看下面的示例，它尝试打开一个文件并打印其内容到屏幕。

```
for line in open("myfile.txt"):
    print line,
```

这段代码的问题就是代码执行完之后它还会让文件在一段不确定的时间内保持打开状态。这在简单的脚本中没什么，但是在大型应用程序中可能是一个问题。With语句可以确保像文件这样的对象总能及时准确地被清理掉。

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

执行该语句后，文件*f* 将始终被关闭，即使在处理某一行时遇到了问题。其它对象是否提供了预定义的清理行为要查看它们的文档。

9. 类

与其他编程语言相比，Python 的类机制用最少的语法和语义引入了类。它是 C++ 和 Modula-3 类机制的混合。Python 的类提供了面向对象编程的所有标准功能： 类继承机制允许有多个基类，继承的类可以覆盖其基类或类的任何方法，方法能够以相同的名称调用基类中的方法。对象可以包含任意数量和种类的数据。和模块一样，类同样具有 Python 的动态性质：它们在运行时创建，并可以在创建之后进一步修改。

用 C++ 术语来讲，通常情况下类成员（包括数据成员）是公有的（其它情况见下文私有变量和类本地引用），所有的成员函数都是虚 的。与 Modula-3 一样，在成员方法中没有简便的方式引用对象的成员：方法函数的声明用显式的第一个参数表示对象本身，调用时会隐式地引用该对象。与 Smalltalk 一样，类本身也是对象。这给导入类和重命名类提供了语义上的合理性。与 C++ 和 Modula-3 不同，用户可以用内置类型作为基类进行扩展。此外，像 C++ 一样，类实例可以重定义大多数带有特殊语法的内置操作符（算术运算符、下标等）。

（由于没有统一的达成共识的术语，我会偶尔使用 SmallTalk 和 C++ 的术语。我比较喜欢用 Modula-3 的术语，因为比起 C++，Python 的面向对象语法更像它，但是我想很少有读者听说过它。）

9.1. 名称和对象

对象是独立的，多个名字（在多个作用域中）可以绑定到同一个对象。这在其他语言中称为别名。第一次粗略浏览 Python 时经常不会注意到这个特性，而且处理不可变的基本类型（数字，字符串，元组）时忽略这一点也没什么问题。然而， 在Python 代码涉及可变对象如列表、 字典和大多数其它类型时，别名可能具有意想不到语义效果。这通常有助于优化程序，因为别名的行为在某些方面类似指针。例如，传递一个对象的开销是很小的，因为在实现上只是传递了一个指针；如果函数修改了参数传递的对象，调用者也将看到变化 —— 这就避免了类似 Pascal 中需要两个不同参数的传递机制。

9.2. Python 作用域和命名空间

在介绍类之前，我首先要告诉你一些有关 Python 作用域的的规则。类的定义非常巧妙的运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的工作原理。另外，这一切的知识对于任何高级 Python 程序员都非常有用。

让我们从一些定义开始。

命名空间是从名称到对象的映射。当前命名空间主要是通过 Python 字典实现的，不过通常不会引起任何关注（除了性能方面），它以后也有可能会改变。以下有一些命名空间的例子：内置名称集（包括函数名例如abs()和内置异常的名称）；模块中的全局名称；函数调用中的

局部名称。在某种意义上的一组对象的属性也形成一个命名空间。关于命名空间需要知道的重要一点是不同命名空间的名称绝对没有任何关系；例如，两个不同模块可以都定义函数maximize而不会产生混淆 —— 模块的使用者必须以模块名为前缀引用它们。

顺便说一句，我使用属性 这个词称呼点后面的任何名称 —— 例如，在表达式z.real中，real是z对象的一个属性。严格地说，对模块中的名称的引用是属性引用：在表达式modname.funcname中， modname是一个模块对象，funcname是它的一个属性。在这种情况下，模块的属性和模块中定义的全局名称之间碰巧是直接的映射：它们共享同一命名空间！[1]

属性可以是只读的也可以是可写的。在后一种情况下，可以对属性赋值。模块的属性都是可写的：你可以这样写modname.the_answer=42。可写的属性也可以用del语句删除。例如，delmodname.the_answer将会删除对象modname中的the_answer属性。

各个命名空间创建的时刻是不一样的，且有着不同的生命周期。包含内置名称的命名空间在Python 解释器启动时创建，永远不会被删除。模块的全局命名空间在读入模块定义时创建；通常情况下，模块命名空间也会一直保存到解释器退出。在解释器最外层调用执行的语句，不管是从脚本文件中读入还是来自交互式输入，都被当作模块**main**的一部分，所以它们有它们自己的全局命名空间。(内置名称实际上也存在于一个模块中，这个模块叫**builtin**。)

函数的局部命名空间在函数调用时创建，在函数返回或者引发了一个函数内部没有处理的异常时删除。（实际上，用遗忘来形容到底发生了什么更为贴切。)当然，每个递归调用有它们自己的局部命名空间。

作用域 是 Python 程序中可以直接访问一个命名空间的代码区域。这里的“直接访问”的意思是用没有前缀的引用在命名空间中找到的相应的名称。

虽然作用域的确定是静态地，但它们的使用是动态地。程序执行过程中的任何时候，至少有三个嵌套的作用域，它们的命名空间是可以直接访问的：

- 首先搜索最里面包含局部命名的作用域
- 其次搜索所有调用函数的作用域，从最内层调用函数的作用域开始，它们包含非局部但也非全局的命名
- 倒数第二个搜索的作用域是包含当前模块全局命名的作用域
- 最后搜索的作用域是最外面包含内置命名的命名空间

如果一个命名声明为全局的，那么对它的所有引用和赋值会直接搜索包含这个模块全局命名的作用域。否则，在最里面作用域之外找到的所有变量都是只读的（对这样的变量赋值会在最里面的作用域创建一个新 的局部变量，外部具有相同命名的那个变量不会改变）。

通常情况下，局部作用域引用当前函数的本地命名。函数之外，局部作用域引用的命名空间与全局作用域相同：模块的命名空间。类定义在局部命名空间中创建了另一个命名空间。

认识到作用域是由代码确定的是非常重要的：函数的全局作用域是函数的定义所在的模块的命名空间，与函数调用的位置或者别名无关。另一方面，命名的实际搜索过程是动态的，在运行时确定的——然而，Python 语言也在不断发展，以后有可能会成为静态的“编译”时确定，所以不要依赖动态解析！（事实上，本地变量是已经确定静态。）

Python的一个特别之处在于——如果没有使用global语法——其赋值操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除也是如此：delx只是从局部作用域的命名空间中删除命名x。事实上，所有引入新命名的操作都作用于局部作用域： 特别是import语句和函数定义将模块名或函数绑定于局部作用域。(可以使用 Global 语句将变量引入到全局作用域。)

9.3. 初识类

类引入了少量的新语法、三种新对象类型和一些新语义。

9.3.1. 类定义语法

类定义的最简单形式如下所示：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类的定义就像函数定义（def语句），要先执行才能生效。(你当然可以把它放进if语句的某一分支，或者一个函数的内部。)

实际应用中，类定义包含的语句通常是函数定义，不过其它语句也是可以的而且有时还会很有用——后面我们会再回来讨论。类中的函数定义通常有一个特殊形式的参数列表，这是由方法调用的协议决定的——同样后面会解释这些。

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有的赋值成为这个新命名空间的局部变量。特别是这里的函数定义会绑定新函数的名字。

类定义正常结束时，一个类对象也就创建了。基本上它是对类定义创建的命名空间进行了一个包装；我们在下一节将进一步学习类对象的知识。原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名（例子中是ClassName）。

9.3.2. 类对象

类对象支持两种操作：属性引用和实例化。

属性引用 使用和Python中所有的属性引用一样的标准语法： obj.name。有效的属性名称是在该类的命名空间中的类对象被创建时的所有名称。因此，如果类定义看起来像这样：

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

那么 MyClass.i 和 MyClass.f 是有效的属性引用，分别返回一个整数和一个方法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。**doc** 也是一个有效的属性，返回类的文档字符串： "Asimpleexampleclass"。

类的实例化 使用函数的符号。可以假设类对象是一个不带参数的函数，该函数返回这个类的一个新的实例。例如（假设沿用上面的类）：

```
x = MyClass()
```

创建这个类的一个新实例，并将该对象赋给局部变量x。

实例化操作（“调用”一个类对象）将创建一个空对象。很多类希望创建的对象可以自定义一个初始状态。因此类可以定义一个名为init()的特殊方法，像下面这样：

```
def __init__(self):
    self.data = []
```

当类定义了init()方法，类的实例化会为新创建的类实例自动调用init()。所以在下面的示例中，可以获得一个新的、已初始化的实例：

```
x = MyClass()
```

当然，init()方法可以带有参数，这将带来更大的灵活性。在这种情况下，类实例化操作的参数将传递给init()。例如，

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. 实例对象

现在我们可以用实例对象做什么？实例对象唯一可用的操作就是属性引用。有两种有效的属性名：数据属性和方法。

数据属性相当于 Smalltalk 中的"实例变量"或 C++ 中的"数据成员"。数据属性不需要声明；和局部变量一样，它们会在第一次给它们赋值时生成。例如，如果x是上面创建的MyClass的实例，下面的代码段将打印出值16而不会出现错误：

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

实例属性引用的另一种类型是方法。方法是"属于"一个对象的函数。（在 Python，方法这个术语不只针对类实例：其他对象类型也可以具有方法。例如，列表对象有 append、insert、remove、sort 方法等等。但是在后面的讨论中，除非明确说明，我们提到的方法特指类实例对象的方法。）

实例对象的方法的有效名称依赖于它的类。根据定义，类中所有函数对象的属性定义了其实例中相应的方法。所以在我们的示例中， x.f是一个有效的方法的引用，因为MyClass.f是一个函数，但x.i不是，因为MyClass.i不是一个函数。但x.f与MyClass.f也不是一回事 —— 它是一个方法对象，不是一个函数对象。

9.3.4. 方法对象

通常情况下，方法在绑定之后被直接调用：

```
x.f()
```

在MyClass的示例中，这将返回字符串'helloworld'。然而，也不是一定要直接调用方法： x.f是一个方法对象，可以存储起来以后调用。例如：

```
xf = x.f
while True:
    print xf()
```

会不断地打印helloworld。

调用方法时到底发生了什么？你可能已经注意到，上面x.f()的调用没有参数，即使f ()函数的定义指定了一个参数。该参数发生了什么问题？当然如果函数调用中缺少参数 Python 会抛出异常——即使这个参数实际上没有使用……

实际上，你可能已经猜到了答案：方法的特别之处在于实例对象被作为函数的第一个参数传给了函数。在我们的示例中，调用x.f()完全等同于MyClass.f(x)。一般情况下，以*n* 个参数的列表调用一个方法就相当于将方法所属的对象插入到列表的第一个参数的前面，然后以新的列表调用相应的函数。

如果你还是不明白方法的工作原理，了解一下它的实现或许有帮助。引用非数据属性的实例属性时，会搜索它的类。如果这个命名确认为一个有效的函数对象类属性，就会将实例对象和函数对象封装进一个抽象对象：这就是方法对象。以一个参数列表调用方法对象时，它被重新拆封，用实例对象和原始的参数列表构造一个新的参数列表，然后函数对象调用这个新的参数列表。

9.3.5. 类和实例变量

一般来说，实例变量用于对每一个实例都是唯一的数据，类变量用于类的所有实例共享的属性和方法：

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

正如在名称和对象讨论的，可变对象，例如列表和字典，的共享数据可能带来意外的效果。例如，下面代码中的*tricks* 列表不应该用作类变量，因为所有的*Dog* 实例将共享同一个列表：

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

这个类的正确设计应该使用一个实例变量：

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. 补充说明

数据属性会覆盖同名的方法属性；为了避免意外的命名冲突，这在大型程序中可能带来极难发现的 bug，使用一些约定来减少冲突的机会是明智的。可能的约定包括大写方法名称的首字母，使用一个小写的独特字符串（也许只是一个下划线）作为数据属性名称的前缀，或者方法使用动词而数据属性使用名词。

数据属性可以被方法引用，也可以由一个对象的普通用户（“客户端”）使用。换句话说，类是不能用来实现纯抽象数据类型。事实上，Python 中不可能强制隐藏数据——那全部基于约定。（另一方面，如果需要，使用 C 编写的 Python 实现可以完全隐藏实现细节并控制对象

的访问；这可以用来通过 C 语言扩展 Python。）

客户应该谨慎的使用数据属性——客户可能通过践踏他们的数据属性而使那些由方法维护的常量变得混乱。注意：只要能避免冲突，客户可以向一个实例对象添加他们自己的数据属性，而不会影响方法的正确性——再次强调，命名约定可以避免很多麻烦。

从方法内部引用数据属性（或其他方法）并没有快捷方式。我觉得这实际上增加了方法的可读性：当浏览一个方法时，在局部变量和实例变量之间不会出现令人费解的情况。

通常，方法的第一个参数称为self。这仅仅是一个约定：名字self对 Python 而言绝对没有任何特殊含义。但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些类 查看 器程序也可能是遵循此约定编写的。

类属性的任何函数对象都为那个类的实例定义了一个方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在f、 g和h都是类C中引用函数对象的属性，因此它们都是C的实例的方法 —— h完全等同于g。请注意，这种做法通常只会混淆程序的读者。

通过使用self参数的方法属性，方法可以调用其他方法：

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以像普通函数那样引用全局命名。与方法关联的全局作用域是包含类定义的模块。（类本身永远不会做为全局作用域使用。）尽管很少有好的理由在方法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用导入全局作用域的函数和模块，也可以调用定义在其中的类和函数。通常，包含此方法的类也会定义在这个全局作用域，在下一节我们会了解为何一个方法要引用自己的类。

每个值都是一个对象，因此每个值都有一个类（也称为类型）。它存储为object.**class**。

9.5. 继承

当然，一个语言特性不支持继承是配不上“类”这个名字的。派生类定义的语法如下所示：

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

BaseClassName必须与派生类定义在一个作用域内。用其他任意表达式代替基类的名称也是允许的。这可以是有用的，例如，当基类定义在另一个模块中时：

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程和基类是相同的。类对象创建后，基类会被保存。这用于解析属性的引用：如果在类中找不到请求的属性，搜索会在基类中继续。如果基类本身是由别的类派生而来，这个规则会递归应用。

派生类的实例化没有什么特殊之处：DerivedClassName()创建类的一个新的实例。方法的引用按如下规则解析： 搜索对应的类的属性，必要时沿基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

派生的类可能重写其基类的方法。因为方法调用本对象中的其它方法时没有特权，基类的方法调用本基类的方法时，可能实际上最终调用了派生类中的覆盖方法。（对于 C++ 程序员：Python 中的所有方法实际上都是虚的。）

派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一个简单的方法可以直接调用基类方法：只要调用BaseClassName.methodname(self,arguments)。有时这对于客户端也很有用。（要注意只有BaseClassName在同一全局作用域定义或导入时才能这样用。）

Python 有两个用于继承的函数：

- 使用isinstance()来检查实例类型：isinstance(obj, int)只有obj.**class**是int或者是从int派生的类时才为True。
- 使用issubclass()来检查类的继承: issubclass(bool，int)是True因为bool是int的子类。然而， issubclass （unicode，str)是False因为unicode不是str的一个子类（它们只是共享一个共同的祖先， basestring） 。

9.5.1. 多继承

Python 也支持一定限度的多继承形式。具有多个基类的类定义如下所示：

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

对于旧风格的类，唯一的规则是深度优先，从左到右。因此，如果在DerivedClassName中找不到属性，它搜索Base1，然后（递归）基类中的Base1，只有没有找到，它才会搜索base2，依此类推。

（对某些人，广度优先——在搜索Base1的基类之前先搜索base2和Base3——看起来更自然。然而，在你能弄明白与base2中的一个属性名称冲突的后果之前，你需要知道Base1的某个特定属性实际上是定义在Base1的还是在其某个基类中的。深度优先规则使Base1的直接属性和继承的属性之间没有差别）。

对于新风格的类，方法的解析顺序动态变化地支持合作对super()的调用。这种方法在某些其它多继承的语言中也有并叫做call-next-method，它比单继承语言中的super调用更强大。

对于新风格的类，动态调整顺序是必要的，因为所有的多继承都会有一个或多个菱形关系(从最底部的类向上，至少会有一个父类可以通过多条路径访问到）。例如，所有新风格的类都继承自object，所以任何多继承都会有多条路径到达object。为了防止基类被重复访问，动态算法线性化搜索顺序，每个类都按从左到右的顺序特别指定了顺序，每个父类只调用一次，这是单调的（也就是说一个类被继承时不会影响它祖先的次序）。所有这些特性使得设计可靠并且可扩展的多继承类成为可能。有关详细信息，请参阅http://www.python.org/download/releases/2.3/mro/。

9.6. 私有变量和类本地引用

在 Python 中不存在只能从对象内部访问的“私有”实例变量。然而，有一项大多数 Python 代码都遵循的公约：带有下划线（例如_spam）前缀的名称应被视为非公开的 API 的一部分（无论是函数、 方法还是数据成员）。它应该被当做一个实现细节，将来如果有变化孰不另行通知。

因为有一个合理的类私有成员的使用场景（即为了避免名称与子类定义的名称冲突），Python 对这种机制有简单的支持，叫做*name mangling*。**spam** 形式的任何标识符(前面至少两个下划线，后面至多一个下划线）将被替换为**_classname**spam，classname是当前类的名字。此mangling会生效而不考虑该标识符的句法位置，只要它出现在类的定义的范围内。

Name mangling 有利于子类重写父类的方法而不会破坏类内部的方法调用。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

请注意 mangling 规则的目的主要是避免发生意外；访问或者修改私有变量仍然是可能的。这在特殊情况下，例如调试的时候，还是有用的。

请注意传递给exec、 eval()或execfile()的代码没有考虑要将调用类的类名当作当前类；这类似于global语句的效果，影响只限于一起进行字节编译的代码。相同的限制适用于getattr()、setattr()和delattr()，以及直接引用**dict**时。

9.7. 零碎的说明

有时候类似于Pascal 的"record" 或 C 的"struct"的数据类型很有用，它们把几个已命名的数据项目绑定在一起。一个空的类定义可以很好地做到：

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

某一段 Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类来模拟该数据类型的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有read ()和readline () 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

实例的方法对象也有属性：m.im_self是具有方法m()的实例对象，m.im_func是方法的函数对象。

9.8. 异常也是类

用户定义的异常类也由类标识。利用这个机制可以创建可扩展的异常层次。

raise语句有两种新的有效的（语义上的）形式：

```
raise Class, instance

raise instance
```

第一种形式中，instance必须是class或者它的子类的实例。第二种形式是一种简写：

```
raise instance.__class__, instance
```

except子句中的类如果与异常是同一个类或者是其基类，那么它们就是相容的（但是反过来是不行的——except子句列出的子类与基类是不相容的）。例如，下面的代码将按该顺序打印 B、 C、 D：

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

请注意，如果except 子句的顺序倒过来 (excpet BB在最前面）， 它就会打印B，B，B ——第一个匹配的异常被触发。

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后是用内置函数str()将类转换得到的完整字符串。

9.9. 迭代器

现在你可能注意到大多数容器对象都可以用for遍历：

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

这种风格的访问明确、 简洁和方便。迭代器的用法在 Python 中普遍而且统一。在后台， for 语句调用容器对象上的iter() 。该函数返回一个定义了next ()方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时， next ()会引发StopIteration异常，告诉for循环终止。此示例显示它是如何工作：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

看过迭代器协议背后的机制后，将很容易将迭代器的行为添加到你的类中。定义一个**iter**()方法，它使用next()方法返回一个对象。如果类定义了next()， **iter**() 可以只返回self：

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

9.10. 生成器

生成器是创建迭代器的一种简单而强大的工具。它们写起来就像是正规的函数，只是需要返回数据的时候使用yield语句。每次next()调用时，生成器再恢复它离开的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了生成器可以非常简单地创建出来：

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

生成器能做到的什么事，前一节所述的基于类的迭代器也能做到。生成器这么紧凑的原因是因为**iter()**和next ()方法是自动创建的。

另一个关键功能是调用时自动保存的本地变量和执行状态。这使得该函数相比实例变量，如self.index和self.data方法，更容易写，更清楚地使用。

除了自动方法创建和保存的程序状态外，当创建完成，他们会自动抛出StopIteration。组合起来，这些功能可以容易地创建迭代器如同编写正规函数。

9.11. 生成器表达式

使用类似列表表示式的语法，一些简单的生成器可以写成简洁的表达式，但是使用圆括号代替方括号。这些表达式用于生成器在封闭的函数中使用的情况。生成器表达式更紧凑但没有完整的生成器定义用途广泛，比等同的列表表示式消耗较少的内存。

例子：

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

脚注

| [1] | 有一件事除外。模块对象具有一个隐藏的只读属性叫做**dict**，它返回用于实现模块命名空间的字典；名称**dict**是一个属性而不是一个全局的名称。很明显，使用它违反了命名空间实现的抽象，应该限制在类似事后调试这样的事情上。 | |-----|-----|

10. 标准库概览

10.1. 操作系统接口

os模块提供了几十个函数与操作系统交互：

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

一定要使用importos的形式而不要用fromosimport*。这将避免os.open()屏蔽内置的open()函数，它们的功能完全不同。

内置的dir()和help()函数对于使用像os大型模块可以作为非常有用的交互式帮助：

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

对于日常的文件和目录管理任务，shutil模块提供了一个易于使用的高级接口：

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2. 文件通配符

glob模块提供了一个函数用于在目录中以通配符搜索文件，并生成匹配的文件列表：

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. 命令行参数

常见的实用程序脚本通常需要处理命令行参数。这些参数以一个列表存储在sys模块的*argv* 属性中。例如下面的输出结果来自于从命令行运行pythondemo.pyone two three：

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

getopt模块使用Unix getopt()函数的约定处理*sys.argv*。argparse模块提供更强大、 更灵活的命令行处理功能。

10.4. 错误输出重定向和程序终止

sys模块还具有*stdin*、*stdout*和*stderr*属性。即使在*stdout*被重定向时，后者也可以用于显示警告和错误信息：

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

终止脚本最直接的方法来是使用sys.exit()。

10.5. 字符串模式匹配

re模块为高级的字符串处理提供了正则表达式工具。对于复杂的匹配和操作，正则表达式提供了简洁、优化的解决方案：

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

当只需要简单的功能时，最好使用字符串方法，因为它们更容易阅读和调试：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. 数学

math模块为浮点运算提供了对底层 C 函数库的访问：

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random模块提供了进行随机选择的工具：

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

10.7. 互联网访问

有很多的模块用于访问互联网和处理的互联网协议。最简单的两个是从URL获取数据的urllib2和发送邮件的smtplib：

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

（请注意第二个示例需要在本地主机上运行邮件服务器）。

10.8. 日期和时间

datetime模块提供了处理日期和时间的类，既有简单的方法也有复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持处理时区。

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. 数据压缩

常见的数据打包和压缩格式都直接支持，这些模块包括：zlib、gzip、bz2、zipfile和tarfile。

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. 性能度量

一些 Python 用户对同一问题的不同解决方法之间的性能差异深有兴趣。Python 提供了的一个度量工具可以立即解决这些问题。

例如，使用元组封装和拆封功能而不是传统的方法来交换参数可能会更吸引人。timeit模块快速证明了现代的方法更快一些：

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

与timeit的精细的粒度相反，profile和pstats模块提供了针对更大代码块的时间度量工具。

10.11. 质量控制

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。

doctest模块提供一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试构造与剪切一个典型的调用并同它的结果粘贴到文档字符串中一样简单。通过用户提供的例子，它发展了文档，允许 doctest 模块确认代码的结果是否与文档一致：

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

unittest模块不像doctest模块那样容易，不过它可以在一个独立的文件里提供一个更全面的测试集：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

10.12. Batteries Included

Python 有"Batteries Included"的哲学。这最好是通过其较大的文件包的先进和强大功能。例如：

- Xmlrpclib和SimpleXMLRPCServer模块让远程过程调用变得轻而易举。尽管模块有这样的名字，它不需要直接XML知识或处理 XML 。
- email包是是一个处理电子邮件的库，包括MIME和其它基于RFC 2822的邮件。与smtplib和poplib用于实际发送和接收邮件，email包有一个完整的工具集用于构建或者解码复杂

邮件结构（包括附件），并实现互联网编码和头协议。

- xml.dom和xml.sax的包为这种流行的数据交换格式提供了强大的支持。同样，csv模块支持以常见的数据库格式直接读取和写入。这些模块和包一起大大简化了 Python 应用程序和其他工具之间的数据交换。
- 国际化支持模块包括gettext、locale和codecs包。

11. 标准库概览 — 第II部分

第二部分包含了更高级的模块，它们支持专业编程的需要。这些模块很少出现在小型的脚本里。

11.1. 输出格式

repr模块提供的repr()的自定义的缩写显示大型或深层嵌套容器的版本：

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

pprint模块提供更复杂的打印控制，以解释器可读的方式打印出内置对象和用户定义的对象。当结果超过一行时，这个"漂亮的打印机"将添加分行符和缩进，以更清楚地显示数据结构：

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap模块格式化文本段落以适应设定的屏宽：

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale模块会访问区域性特定数据格式的数据库。分组属性的区域设置的格式函数的格式设置的数字以直接的方式提供了组分隔符：

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. 模板

string模块包括一个通用Template类，它用简化的语法适合最终用户编辑。这允许用户自定义他们的应用程序无需修改应用程序。

这种格式使用的占位符名称由$与有效的 Python 标识符（字母数字字符和下划线）组成。周围的大括号与占位符允许它应遵循的更多字母数字字母并且中间没有空格。$$创建一个转义的$：

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

当字典或关键字参数中没有提供占位符时，substitute()方法将引发KeyError。对于邮件-合并风格的应用程序，用户提供的数据可能不完整，这时safe_substitute()方法可能会更合适 —— 如果没有数据它将保持占位符不变：

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 类的子类可以指定自定义的分隔符。例如，图像浏览器的批量命名工具可能选用百分号作为表示当前日期、图像 序列号或文件格式的占位符：

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '{0} --> {1}'.format(filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

模板的另一个应用是把多样的输出格式细节从程序逻辑中分类出来。这使它能够替代用户的 XML 文件、 纯文本报告和 HTML 网页报表。

11.3. 二进制数据记录格式

The struct module provides pack() and unpack() functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the zipfile module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # skip to the next header
```

11.4. 多线程

线程是一种解耦非顺序依赖任务的技术。线程可以用来提高接应用程序受用户输入的响应速度，而其他任务同时在后台运行。一个相关的使用场景是 I/O 操作与另一个线程中的计算并行执行。

下面的代码演示在主程序连续运行的同时，threading模块如何在后台运行任务：

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

多线程应用程序的最主要挑战是协调线程间共享的数据或其他资源。为此目的，该线程模块提供了许多同步原语包括锁、 事件、 条件变量和信号量。

尽管这些工具很强大，很小的设计错误也可能导致很难复现的问题。因此，任务协调的首选方法是把对一个资源的所有访问集中在一个单独的线程中，然后使用Queue模块用那个线程服务其他线程的请求。应用程序使用Queue.Queue对象进行线程间的通信和协调将更容易设计、 更具可读性和更可靠。

11.5. 日志

logging模块提供了一个具有完整功能并且非常灵活的日志系统。最简单的，发送消息到一个文件或者sys.stderr：

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

这将生成以下输出：

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

默认情况下，信息和调试消息被压制并输出到标准错误。其他输出选项包括将消息通过 email、 datagrams、sockets发送，或者发送到 HTTP 服务器。根据消息的优先级，新的过滤器可以选择不同的方式：DEBUG、INFO、WARNING、ERROR和CRITICAL。

日志系统可以直接在 Python 代码中定制，也可以不经过应用程序直接在一个用户可编辑的配置文件中加载。

11.6. 弱引用

Python 会自动进行内存管理 (对大多数的对象进行引用计数和垃圾回收以循环利用）。在最后一个引用消失后，内存会立即释放。

这个方式对大多数应用程序工作良好，但是有时候会需要跟踪对象，只要它们还被其它地方所使用。不幸的是，只是跟踪它们也会创建一个引用，这将使它们永久保留。weakref模块提供工具用来跟踪对象而无需创建一个引用。当不再需要该对象时，它会自动从 weakref 表中删除并且会为 weakref 对象触发一个回调。典型的应用包括缓存创建的时候需要很大开销的对象：

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 列表工具

很多数据结构使用内置列表类型就可以满足需求。然而，有时需要其它具有不同性能的替代实现。

The array module provides an array() object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections模块提供了一个deque()对象，就像一个列表,不过它从左边添加和弹出更快，但是在内部查询更慢。这些对象非常适合实现队列和广度优先的树搜索：

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

除了列表的替代实现，该库还提供了其它工具例如bisect模块中包含处理排好序的列表的函数：

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq模块提供的函数可以实现基于常规列表的堆。最小的值总是保持在第零个位置。这对循环访问最小元素，但是不想运行完整列表排序的应用非常有用：

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8. 十进制浮点数运算

decimal模块提供一个Decimal数据类型用于为十进制浮点运算。相比二进制浮点数内置的float实现，这个类对于以下情形特别有用：

- 财务应用程序和其他用途，需要精确的十进制表示形式，
- 控制精度，
- 对符合法律或法规要求，舍入的控制
- 跟踪有效小数位
- 用户希望计算结果与手工计算相符的应用程序。

例如，计算上 70%电话费的 5%税给不同的十进制浮点和二进制浮点结果。区别变得重要如果结果舍入到最接近的分:

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

Decimal的结果总是保有结尾的0，自动从两位精度延伸到4位。Decimal 类似手工完成的数学运算，这就避免了二进制浮点数无法精确表达数据精度产生的问题。

精确地表示允许Decimal可以执行二进制浮点数无法进行的模运算和等值测试：

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

decimal模块提供任意精度的运算：

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12.现在怎么办?

阅读本教程可能让你对使用 Python 更感兴趣了——你应该会渴望将 Python 应用于解决实际问题。你应该到哪里去了解更多 Python 的内容呢？

本教程是 Python 文档集的一部分。文档集中的一些其它文件有：

- *Python 标准库*：

你应该浏览本手册，它给出了标准库中关于类型、 函数和模块的完整（虽然简洁）的参考资料。标准的 Python 发布包含大量 的附加模块。其中有读取 Unix 邮箱、收取HTTP 文档、生成随机数、解析命令行选项、编写 CGI 程序、压缩数据以及很多其它任务的模块。浏览一下这个库参考手册会让你知道有什么是现成可用的。

- *安装 Python 模块* 解释如何安装由其他 Python 用户编写的外部模块。
- *Python 语言参考*：详细地讲述了 Python 的语法和语义。它读起来很难，但是作为语言本身的完整指南非常有用。

更多的 Python 资源：

- http://www.python.org：主要的 Python Web 站点。它包含代码、文档和网上 Python 相关页面的链接。该网站在世界各地都有镜像，如欧洲、 日本和澳大利亚；镜像可能会比主站快，这取决于你的地理位置。
- http://docs.python.org：快速访问 Python 的文档。
- http://pypi.python.org： Python 包索引，以前的绰号叫奶酪店，是用户创建的 Python 模块的索引，这些模块可供下载。一旦你开始发布代码，你可以在这里注册你的代码这样其他人可以找到它。
- http://aspn.activestate.com/ASPN/Python/Cookbook/: 这本 Python 食谱收集了相当多的代码示例、大型的模块，以及有用的脚本。其中尤其显著的贡献被收集成一书，这本书也叫做 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3)。

Python 相关的问题和问题报告，你可以发布到新闻组*comp.lang.python*，或将它们发送到邮件列表python-list@python.组织结构图。新闻组和邮件列表是互通的，因此发布到其中的一个消息将自动转发给另外一个。一天大约有 120 个帖子（最高峰到好几百)，包括询问（和回答）问题，建议新的功能和宣布新的模块。在发帖之前，一定要检查常见问题（也称为 FAQ）的列表。可在http://mail.python.org/pipermail/查看邮件列表的归档。FAQ 回答了很多经常出现的问题，可能已经包含你的问题的解决方法。

13. 交互式输入的编辑和历史记录

某些版本的 Python 解释器支持编辑当前的输入行和历史记录，类似于在 Korn shell 和 GNU Bash shell 中看到的功能。这是使用GNU Readline库实现的，它支持 Emacs 风格和 vi 风格的编辑。这个库有它自己的文档，在这里我不就重复了；然而，基本原理很容易解释。本章讲述的交互式编辑和历史记录功能在 Unix 版本和 Cygwin 版本中是可选的。

本章不 是Mark Hammond的 PythonWin 包或者随 Python 一起发布的基于 TK 的 IDLE 环境的文档。基于 NT 系统的DOS和其它 DOS、Windows 系统上的命令行历史回溯是另一个话题。

13.1. 行编辑

如果支持，无论解释器打印主提示符还是从属提示符，输入行一直都可以编辑。可以使用传统的Emacs控制字符编辑当前行。其中最重要的是：C-A (Control-A)将光标移动到行首，C-E移到行尾，C-B将光标向左移动一个位置，C-F向右移动一个位置。退格键删除光标左侧的字符，C-D删除光标右侧的字符。C-k删掉一行中光标右侧的所有字符，C-y将最后一次删除的字符串粘贴到光标位置。C-下划线将撤消最近一次的更改；它可以重复执行并产生累积效果。

13.2. 历史记录

历史记录的工作方式如下。所有的非空输入行都保存在历史记录缓冲区中，当给出一个新的提示符时，你位于缓冲区最底部新的一行。C-P向上（往回）移动历史记录缓冲区中的一行，C-N向下移动一行。历史记录缓冲区中的任何一行都可以编辑；提示符的前面用一个星号标记修改的行。按Return键会将当前行传递给解释器。C-R开始逐渐向后搜索；C-S开始向前搜索。

13.3. 快捷键绑定

通过将命令放在一个名为~/.inputrc的初始化文件中，可以自定义快捷键和Readline 库的一些其他参数。快捷键绑定有如下方式

```
key-name: function-name
```

或

```
"string": function-name
```

也可以设置选项

```
set option-name value
```

例如：

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

注意 Python 中Tab键默认绑定的是插入一个制表符而不是 Readline 的默认文件名补全功能。如果你坚持这样做，你可以放置一行

```
Tab: complete
```

在你的~/.inputrc中。（当然，如果你习惯使用Tab作为缩进，这会使得输入续行的缩进很困难。）

可以选择变量和模块名自动补全功能。若要在解释器的交互模式中启用这个功能，添加以下内容到你的启动文件：[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

这将Tab键绑定到补全功能，所以按Tab键两次将会给出补全的建议；它会查找 Python 语句名称、当前的局部变量以及有效的模块名。例如点分表达式string.a，它会先解析最后一个点'.'之前的表达式，然后根据结果对象给出建议补全的内容。注意如果表达式得到的对象带有**getattr()**方法，这可能会执行应用程序定义的代码。

更有用的启动文件可能看起来像下面这个示例。注意下面的示例删除了它创建的名称，一旦这些名称不再需要；这是因为启动文件与交互式命令在相同的命名空间中执行，删除这些名称可以避免在交互式环境中产生副作用。你可能发现保留一些导入的模块有时也会很方便，例如os，因为在大多数与解释器的交互中都需要它。

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4. 其它交互式解释器

与早期版本的解释器相比，现在是向前巨大的进步；然而，有些愿望还是没有实现：如果能对连续的行给出正确的建议就更好了（解析器知道下一行是否需要缩进）。补全机制可以使用解释器的符号表。检查（或者只是建议）匹配的括号、 引号的命令等也会非常有用。

一个增强的交互式解释器是IPython，它已经存在相当一段时间，具有tab补全、 对象 exploration 和高级的历史记录功能。它也可以彻底定制并嵌入到其他应用程序中。另一个类似的增强的交互式环境是bpython。

脚注

| [1] | 当你启动一个交互式解释器时，Python将执行环境变量PYTHONSTARTUP指定的文件中的内容。若还要定制化非交互式的Python，请参阅定制化模块。 | |-----|-----|

14. 浮点数运算：问题和局限

浮点数在计算机硬件中表示为以 2 为底（二进制）的小数。例如，十进制小数

```
0.125
```

是1/10 + 2/100 + 5/1000 的值，同样二进制小数

```
0.001
```

是 0/2 + 0/4 + 1/8 的值。这两个小数具有相同的值，唯一真正的区别是，第一个小数是十进制表示法，第二个是二进制表示法。

不幸的是，大多数十进制小数不能完全用二进制小数表示。结果是，一般情况下，你输入的十进制浮点数仅由实际存储在计算机中的近似的二进制浮点数表示。

这个问题在十进制情况下很容易理解。考虑分数 1/3，你可以用十进制小数近似它：

```
0.3
```

或者更接近的

```
0.33
```

或者再接近一点的

```
0.333
```

等等。无论你愿意写多少位数字，结果永远不会是精确的 1/3，但将会越来越好地逼近 1/3。

同样地，无论你使用多少位的二进制数，都无法确切地表示十进制值 0.1。1/10 用二进制表示是一个无限循环的小数。

```
0.0001100110011001100110011001100110011001100110011...
```

在任何有限数量的位停下来，你得到的都是近似值。

在一台运行 Python 的典型机器上, Python 浮点数具有 53 位的精度，所以你输入的十进制数 0.1存储在内部的是二进制小数

```
0.0001100110011001100110011001100110011001100110011001101 0
```

非常接近，但不完全等于 1/10。

由于解释器显示浮点数的方式，很容易忘记存储在计算机中的值是原始的十进制小数的近似。Python 只打印机器中存储的二进制值的十进制近似值。如果 Python 要打印 0.1 存储的二进制的真正近似值，将会显示

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这么多位的数字对大多数人是没有用的，所以 Python 显示一个舍入的值

```
>>> 0.1
0.1
```

意识到在真正意义上这是一种错觉是很重要的：机器中的值不是精确的 1/10，它显示 的只是机器中真实值的舍入。一旦你用下面的数值进行算术运算，这个事实就变得很明显了

```
>>> 0.1 + 0.2
0.30000000000000004
```

注意，这是二进制浮点数的自然性质：它不是 Python 中的一个 bug，也不是你的代码中的 bug。你会看到所有支持硬件浮点数算法的语言都会有这个现象（尽管有些语言默认情况下或者在所有输出模式下可能不会显示 出差异）。

还有其它意想不到的。例如，如果你舍入2.675到两位小数，你得到的是

```
>>> round(2.675, 2)
2.67
```

内置round()函数的文档说它舍入到最接近的值，rounding ties away from zero。因为小数 2.675 正好是 2.67 和 2.68 的中间，你可能期望这里的结果是（二进制近似为）2.68。但是不是的，因为当十进制字符串2.675转换为一个二进制浮点数时，它仍然被替换为一个二进制的近似值，其确切的值是

```
2.67499999999999982236431605997495353221893310546875
```

因为这个近似值稍微接近 2.67 而不是 2.68，所以向下舍入。

如果你的情况需要考虑十进制的中位数是如何被舍入的，你应该考虑使用decimal模块。顺便说一下，decimal模块还提供了很好的方式可以“看到”任何 Python 浮点数的精确值。

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

另一个结果是，因为 0.1 不是精确的 1/10，十个值为 0.1 数相加可能也不会正好是 1.0：

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

二进制浮点数计算有很多这样意想不到的结果。“0.1”的问题在下面"误差的表示"一节中有准确详细的解释。更完整的常见怪异现象请参见浮点数的危险。

最后我要说，“没有简单的答案”。也不要过分小心浮点数！Python 浮点数计算中的误差源之于浮点数硬件，大多数机器上每次计算误差不超过 2**53 分之一。对于大多数任务这已经足够了，但是你要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的舍入错误。

虽然确实有问题存在，对于大多数平常的浮点数运算，你只要简单地将最终显示的结果舍入到你期望的十进制位数，你就会得到你期望的最终结果。关于如何精确控制浮点数的显示请参阅格式化字符串的语法中str.format()方法的格式说明符。

14.1. 二进制表示的误差

这一节将详细解释“0.1”那个示例，并向你演示对于类似的情况自已如何做一个精确的分析。假设你已经基本了解浮点数的二进制表示。

二进制表示的误差 指的是这一事实，一些（实际上是大多数) 十进制小数不能精确地用二进制小数表示。这是为什么 Python（或者 Perl、 C、 C++、 Java、 Fortran和其他许多语言）通常不会显示你期望的精确的十进制数的主要原因：

```
>>> 0.1 + 0.2
0.30000000000000004
```

这是为什么？1/10 和 2/10 不能用二进制小数精确表示。今天（2010 年 7 月）几乎所有的机器都使用 IEEE-754 浮点数算法，几乎所有的平台都将 Python 的浮点数映射成 IEEE-754“双精度浮点数”。754 双精度浮点数包含 53 位的精度，所以输入时计算机努力将 0.1 转换为最接

近的 *J*/2***N* 形式的小数，其中*J* 是一个 53 位的整数。改写

```
1 / 10 ~= J / (2**N)
```

为

```
J ~= 2**N / 10
```

回想一下 *J* 有 53 位（>=2**52**但**<2**53），所以 *N* 的最佳值是 56：

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

即 56 是 *N* 保证 *J* 具有 53 位精度的唯一可能的值。*J* 可能的最佳值是商的舍入：

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

由于余数大于 10 的一半，最佳的近似值是向上舍入：

```
>>> q+1
7205759403792794
```

因此在 754 双精度下 1/10 的最佳近似是 J 取大于 2**56 的那个数，即

```
7205759403792794 / 72057594037927936
```

请注意由于我们向上舍入，这其实有点大于 1/10；如果我们没有向上舍入，商数就会有点小于 1/10。但在任何情况下它都不可能是精确的 1/10!

所以计算机从来没有"看到"1/10： 它看到的是上面给出的精确的小数，754 双精度下可以获得的最佳的近似了：

```
>>> .1 * 2**56
7205759403792794.0
```

如果我们把这小数乘以 10**30，我们可以看到其（截断后的）值的最大 30 位的十进制数：

```
>>> 7205759403792794 * 10**30 // 2**56
100000000000000005551115123125L
```

也就是说存储在计算机中的精确数字约等于十进制值 0.100000000000000005551115123125。以前 Python 2.7 和 Python 3.1 版本中，Python 四舍五入到 17 个有效位，给出的值是 '0.10000000000000001'。在当前版本中，Python 显示一个最短的十进制小数，它会正确舍入真实的二进制值，结果就是简单的‘0.1’。

Python 2 标准库

*Python语言参考*讲述Python语言准确的语法和语义，而该库参考手册讲述与Python一起发布的标准库。它还讲述在Python发布中某些常见的可选组件。

Python的标准库非常广泛，它们提供范围很广的工具，下面列出的长长的目录可以表明。这个库包含提供访问系统功能的内建模块（以C语言编写），例如文件I/O，否则其对于Python程序员将是无法访问的，同时它还包含Python语言编写的模块，可为日常编程出现的许多问题提供标准化的方案。其中某些模块明确地为鼓励和增强Python程序的可移植性而设计，通过将平台相关抽象成平台无关的API。

Windows平台上的Python安装程序通常包含完整的标准库并经常包含很多额外的组件。对于Unix类操作系统，Python通常以一组包提供，所以可能需要使用操作系统提供的包管理工具来获取部分或者所有的可选组件。

除该标准库之外，还有正在不断增长的几千个组件（从单个程序和模块到包以及完整的应用程序开发框架）可以从Python包索引获得。

1. 引言

“Python标准库”包含几个不同类型的组件。

如数字和列表，一般被认为是 编程语言的核心 数据类型。对这些类型， Python语言内核 定义了 简单的形式 ，对其语法作了一些约束，但不是完全定义它们的语法。(另一方面，语言核心的确定义了一些语法特性，如操作符的拼写和优先级.)

标准库还包括内置的函数和表达式。 即那些可以被直接使用，而不需要 import 语句另外导入的对象。其中一些被核心语言定义，但大部分对核心语法而言并不是必须的，只是在这里介绍一下。

其实 大量的库集合在一起组成了一些模块。可以从许多不同的角度来剖析这些集合。一些模块用C写成，内置在Python解释器中。其他的模块就是Python写的，以Python源码的形式导入。一些模块提供的是针对Python的接口，如打印一个堆栈的信息。一些模块是针对特定的操作系统，像连接特定的硬件的。还有一些模块是针对特定应用领域的，如互联网。许多的模块在全部的Python版本和端口中都可以使用。而另外一些只有在系统支持和需要的情况下才可以使用。还有一些需要在编译和安装Python的时候 配置了相应的选项才可以使用。

这个手册以“由内而外”的方式组织：它首先介绍了内置的数据类型，然后内置的函数和表达式，最后是按照相关性组织把模块组织成一些章节。章节的顺序是按照章节中的模块大致从常用到不那么重要来排列的。

这意味着，你可以从头读这个手册，当你烦了的时候就跳到下一章，这样对于Python标准库都提供了哪些模块，能支持哪些应用能有一个大概了解。当然，也不用像读小说地一样读它，你可以先看看目录（在手册最前面），寻找特定的函数，模块，或者在搜索框（在后面）里面查找一下。最后，如果你享受随机选择一个主题去阅读的方式，你可以先选择一个随机页码，读上一两段。 无论你想怎么读， 还是建议从内置函数 这一章开始读 ，原因是 其他的章节都是在假设 你已经了解 内置函数 这一章的基础上介绍的。

让我们开始吧！

2. 内建函数

Python解释器内置了一些函数，它们总是可用。这里将它们按字母顺序列出。

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	**import**()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

abs(*x*) 返回一个数的绝对值。参数可以是普通的整数，长整数或者浮点数。如果参数是个复数，返回它的模。

all(*iterable*) 如果*iterable*的所有元素为真（或者iterable为空）， 返回True。等同于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

添加于版本2.5。

any(*iterable*) 如果*iterable*的任一元素为真，返回True。如果iterable为空，返回False。等同于：

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

2.5版本添加。

basestring() 这个抽象类型是str和unicode的超类。它不能被调用或者实例化，但是可以用来测试一个对象是否是str或者unicode的实例。isinstance(obj,basestring)等同于isinstance(obj,(str,unicode))。

出现于版本2.3。

bin(*x*) 将一个整数转化成一个二进制字符串。结果是一个合法的Python表达式。如果*x*不是一个Python int对象，它必须定义一个返回整数的**index**()方法。

出现于版本2.6。

bool([*x*]) 将一个值转化成布尔值，使用标准的真值测试例程。如果*x*为假或者被忽略它返回False；否则它返回True。bool也是一个类，它是int的子类。bool不能被继承。它唯一的实例就是False和True。

出现于版本2.2.1。

改变于版本 2.3：如果没有参数，函数返回False。

bytearray([*source*[, *encoding*[, *errors*]]]) 返回一个新的字节数组。bytearray类型是一个可变的整数序列，整数范围为0 <= x < 256（即字节）。它有可变序列的大多数方法，参见*Mutable Sequence Types*，同时它也有str类型的大多数方法，参见*String Methods*。

*source*参数可以以不同的方式来初始化数组，它是可选的：

- 如果是*string*，必须指明*encoding*（以及可选的*errors*）参数；bytearray()使用str.encode()将字符串转化为字节数组。
- 如果是*integer*，生成相应大小的数组，元素初始化为空字节。
- 如果是遵循*buffer*接口的对象，对象的只读buffer被用来初始化字节数组。
- 如果是*iterable*，它的元素必须是整数，其取值范围为0<=x<256，用以初始化字节数组。

如果没有参数，它创建一个大小为0的数组。

出现于版本2.6。

callable(*object*) 如果*object*参数可调用，返回True；否则返回False。如果返回真，对其调用仍有可能失败；但是如果返回假，对*object*的调用总是失败。注意类是可调用的（对类调用返回一个新实例）；如果类实例有**call**()方法，则它们也是可调用的。

chr(*i*) 返回一个单字符字符串，字符的ASCII码为整数*i*。例如，chr(97)返回字符串'a'。它是ord()的逆运算。参数的取值范围为[0..255]的闭区间；如果*i*超出取值范围，抛出ValueError。参见unichr()。

classmethod(*function*) 将*function*包装成类方法。

类方法接受类作为隐式的第一个参数，就像实例方法接受实例作为隐式的第一个参数一样。声明一个类方法，使用这样的惯例：

```
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
        ...
```

@classmethod是函数*decorator*（装饰器）参见*Function definitions*中的函数定义。

它即可以通过类来调用（如C.f()），也可以通过实例来调用（如C().f()）。除了实例的类，实例本身被忽略。如果在子类上调用类方法，子类对象被传递为隐式的第一个参数。

类方法不同于C++或Java中的静态方法。如果你希望静态方法，参见这节的staticmethod()。

需要类方法更多的信息，参见*The standard type hierarchy*中标准类型层次部分的文档。

出现于版本2.2。

改变于版本2.4：添加了函数装饰器语法。

cmp(*x*, *y*) 比较两个对象*x*和*y*，根据结果返回一个整数。如果xy，返回正数。

compile(*source*, *filename*, *mode*[, *flags*[, *dont_inherit*]]) 将*source*编译成代码对象，或者AST（Abstract Syntax Tree，抽象语法树）对象。代码对象可以经由exec语句执行，或者通过调用eval()演算。*source*可以是Unicode字符串，*Latin-1*编码的字符串或者AST对象。参考ast模块文档以了解如何和AST对象一起使用的信息。

*filename*参数指明一个文件，从该文件中读取（源）代码；如果代码不是从文件中读入，传递一个可识别的值（一般用''）。

*mode*参数指明了编译成哪一类的代码；它可以是'exec'，如果*source*包含一组语句；也可以是'eval'，如果是单一的表达式；或者是'single'，如果是单一的交互式语句（对于最后一种情况，如果表达式语句演算成非None，它的值会被打印）。

可选的参数*flags*和*dont_inherit*控制哪些future语句（见**PEP 236**）影响*source*的编译。如果没有这两个参数（或者都为0），使用调用compile的代码当前有效的future语句来编译source。如果给出了*flags*参数且没有给出*dont_inherit*参数（或者为0），除了本该使用的future语句之外，由*flags*参数指明的future语句也会影响编译。如果*dont_inherit*是非0整数，*flags*参数被忽略（调用compile周围的有效的future语句被忽略）。

future语句由bit位指明，这些bit可以做或运算，以指明多个语句。可以在**future**模块中，_Feature实例的compiler_flag属性找到指明功能的bit位。

如果被编译的源代码是不合法的，函数抛出SyntaxError；如果源代码包含空字节，函数抛出TypeError。

注意

当以'single'或者'eval'模式编译多行代码字符串的时候，输入至少以一个新行结尾。这主要是便于code模块检测语句是否结束。

改变于版本 2.3：添加了*flags*和*dont_inherit*参数。

改变于版本2.6：支持编译AST对象。

改变于版本2.7：允许使用Windows和Mac的新行字符。'exec'模式下的输入不需要以新行结束。

complex([*real*[, *imag*]]) 创建一个复数，它的值为*real* + *imag***j*；或者将一个字符串／数字转化成一个复数。如果第一个参数是个字符串，它将被解释成复数，同时函数不能有第二个参数。第二个参数不能是字符串。每个参数必须是数值类型（包括复数）。如果imag*被忽略，它的默认值是0，这时该函数就像是int()，long()和float()这样的数值转换函数。如果两个参数都被忽略，返回0j。

注意

当从字符串转化成复数的时候，字符串中+或者-两边不能有空白。例如，complex('1+2j')是可行的，但complex('1+2j')会抛出ValueError异常。

在*Numeric Types — int, float, long, complex*中有对复数的描述。

delattr(*object*, *name*) 这个函数和setattr()有关。参数是一个对象和一个字符串。字符串必须是对象的某个属性的名字。只要对象允许，这个函数删除该名字对应的属性。例如，delattr(x,'foobar')等同于delx.foobar。

dict(***kwarg***)**dict(*mapping***, *kwarg)*dict(*iterable, **kwarg*)* 创建一个新字典。*dict*对象就是字典类。参见*dict*和*[*Mapping Types — dict*](#)以得到关于该类的文档。

参见list，set，和tuple类以及collections模块了解其它的容器。

dir([*object*]) 如果没有参数，返回当前本地作用域内的名字列表。如果有参数，尝试返回参数所指明对象的合法属性的列表。

如果对象有**dir**()方法，该方法被调用且必须返回一个属性列表。这允许实现了定制化的**getattr**()或者**getattribute**()函数的对象定制dir()报告对象属性的方式。

如果对象没有提供**dir()**，同时如果对象有定义**dict**属性，dir()会先尝试从**dict**属性中收集信息，然后是对象的类型对象。结果列表没有必要是完整的，如果对象有定制化的**getattr()**，结果还有可能是不准确的。

对于不同类型的对象，默认的dir()行为也不同，因为它尝试产生相关的而不是完整的信息：

- 如果对象是模块对象，列表包含模块的属性名。
- 如果对象是类型或者类对象，列表包含类的属性名，及它的基类的属性名。
- 否则，列表包含对象的属性名，它的类的属性名和类的基类的属性名。

返回的列表按字母顺序排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape(object):
        def __dir__(self):
            return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

注意

因为dir()主要是为了在交互式环境下使用方便，它尝试提供有意义的名字的集合，而不是提供严格或一致定义的名字的集合，且在不同的版本中，具体的行为也有所变化。例如，如果参数是一个类，那么元类属性就不会出现在结果中。

divmod(*a*, *b*) 在长整数除法中，传入两个数字（非复数）作为参数，返回商和余数的二元组。对于混合的操作数类型，应用二元算术运算符的规则。对于一般或者长整数，结果等同于(a//b,a%b)。对于浮点数结果是(q,a%b)，*q*一般是math.floor(a/b)，但也可能比那小1。不管怎样，q*b+a%b*非常接近于a，如果*a%b*非*0*，它和b*符号相同且0<=abs(a%b)<abs(b)。

改变于版本2.3：废弃了在divmod()中使用复数。

enumerate(*sequence*, *start=0*) 返回一个枚举对象。*sequence*必须是个序列，迭代器*iterator*，或者支持迭代的对象。enumerate()返回的迭代器的next()方法返回一个元组，它包含一个计数（从*start*开始，默认为0）和从*sequence*中迭代得到的值：

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

等同于：

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

出现于版本2.3。

改变于版本2.6：添加了*start*参数。

eval(*expression*[, *globals*[, *locals*]]) 参数是Unicode或者*Latin-1*编码的字符串，全局变量和局部变量可选。如果有全局变量，*globals*必须是个字典。如果有局部变量，*locals*可以是任何映射类型对象。

改变于版本2.4：在此之前*locals*需要是个字典。

*expression*参数被当作Python表达式来解析并演算（技术上来说，是个条件列表），使用*globals*和*locals*字典作为全局和局部的命名空间。如果*globals*字典存在，且缺少'**builtins**'，在*expression*被解析之前，当前的全局变量被拷贝进*globals*。这意味着一般来说*expression*能完全访问标准**builtin**模块，且受限的环境会传播。如果*locals*字典被忽略，默认是*globals*字典。如果都被忽略，表达式在eval()被调用的环境中执行。返回值是被演算的表达式的结果。语法错误报告成异常。例子：

```
>>> x = 1
>>> print eval('x+1')
2
```

该函数也能执行任意的代码对象（如compile()返回的结果）。在这种情况下，传递代码对象而不是字符串。如果代码对象编译时*mode*参数为'exec'，eval()返回None。

提示：exec语句支持动态的语句执行。execfile()函数支持执行文件中的语句。globals()和locals()函数返回当前的全局变量和局部变量的字典，可以传递给eval()或者execfile()。

参见ast.literal_eval()，该函数能安全演算只含字面量的表达式的字符串。

execfile(*filename*[, *globals*[, *locals*]]) 该函数类似于exec语句，它解析一个文件而不是字符串。它不同于import语句的地方在于它不使用模块管理——它无条件的读入文件且不会创建一个新模块。[1]

参数是个文件名和两个可选的字典。文件被当成Python语句序列来解析并演算（类似于模块），使用*globals*和*locals*字典作为全局和局部的命名空间。如果存在，*locals*可以是任意的映射类型对象。记住在模块级别，全局和局部字典是同一个字典。如果*globals*和*locals*是两个不同的对象，代码就好象是嵌入在类定义中被执行。

改变于版本2.4：在此之前*locals*必须是个字典。

如果*locals*字典被忽略，默认是*globals*字典。如果两个字典都被忽略，表达式在execfile()被调用的环境中执行。返回None。

注意

默认的*locals*的行为和下述的locals()函数一样：不应该尝试修改默认的*locals*字典。如果在execfile()函数返回后，你希望看到作用于*locals*的代码的效果，显示地传递一个*locals*字典。execfile()不能用于可靠地修改一个函数的局部变量。

file(*name*[, *mode*[, *buffering*]]) file类型的构造函数，进一步的描述见*File Objects*章节。构造函数的参数同下述的open()内置函数。

要打开一个文件，建议使用open()而不是直接调用该构造函数。file更适合于类型测试（例如，isinstance(f,file)）。

出现于版本2.2。

filter(*function*, *iterable*) 构造一个列表，列表的元素来自于*iterable*，对于这些元素*function*返回真。*iterable*可以是个序列，支持迭代的容器，或者一个迭代器。如果*iterable*是个字符串或者元组，则结果也是字符串或者元组；否则结果总是列表。如果*function*是None，使用特性函数，即为假的*iterable*被移除。

注意，在function不为None的情况下，filter(function,iterable)等同于[itemforiteminiterableiffunction(item)]；否则等同于[itemforiteminiterableifitem]（function为None）。

参见itertools.ifilter()和itertools.ifilterfalse()，以得到该函数的迭代器版本，以及该函数的变体（过滤*function*返回假的元素）。

float([*x*]) 将字符串或者数字转化成浮点数。如果参数是字符串，它必须包含小数或者浮点数（可以有符号），周围可以有空白。参数也可以是[+|-]nan或者[+|-]inf。其它情况下，参数可以是原始／长整数或者浮点数，（以Python的浮点数精度）返回具有相同值的浮点数。如果没有参数，返回0.0。

注意

当传递字符串时，依赖于底层的C库，可以返回NaN（Not a Number，不是一个数字）和Infinity（无穷大）这样的值。该函数接受字符串nan（NaN），inf（正无穷大）和-inf（负无穷大）。对于NaN，不区分大小写和+/-号。总是用nan，inf或者-inf来表示NaN和Infinity。

float类型描述于*Numeric Types — int, float, long, complex*。

format(*value*[, *format_spec*]) 将*value*转化成“格式化”的表现形式，格式由*format_spec*控制。对*format_spec*的解释依赖于*value*参数的类型，大多数内置类型有标准的格式化语法：*Format Specification Mini-Language*。

注意

format(value,format*spec)*仅仅调用*value._format*(format_spec)。

出现于版本2.6。

frozenset([*iterable*]) 返回一个新的frozenset对象，如果可选参数*iterable*存在，frozenset的元素来自于iterable。frozenset是个内置类。参见frozenset和*Set Types — set, frozenset*。

关于其它容器，参见set，list，tuple，和dict类，以及collections模块。

出现于版本2.4。

getattr(*object*, *name*[, *default*]) 返回*object*的属性值。*name*必须时个字符串。如果字符串时对象某个属性的名字，则返回该属性的值。例如，getattr(x,'foobar')等同于x.foobar。如果名字指明的属性不存在，且有*default*参数，default被返回；否则抛出AttributeError。

globals() 返回表示当前全局符号表的字典。它总是当前模块的字典（在函数或者方法中，它指定义的模块而不是调用的模块）。

hasattr(*object*, *name*) 参数是一个对象和一个字符串。如果字符串是对象某个属性的名字，返回True；否则返回False。（实现方式为调用getattr(object,name)，看它是否抛出异常）。

hash(*object*) 返回对象的hash（哈希/散列）值（如果有的话）。hash值是整数。它被用于在字典查找时快速比较字典的键。相同的数值有相同的hash（尽管它们有不同的类型，比如1和1.0）。

help([*object*]) 调用内置的帮助系统。（这个函数主要用于交互式使用。）如果没有参数，在解释器的控制台启动交互式帮助系统。如果参数是个字符串，该字符串被当作模块名，函数名，类名，方法名，关键字或者文档主题而被查询，在控制台上打印帮助页面。如果参数是其它某种对象，生成关于对象的帮助页面。

这个函数经由site模块加入内置的命名空间。

出现于版本2.2。

hex(*x*) 将任意大小的整数转化成以“0x”打头的小写的十六进制字符串，例如：

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
>>> hex(1L)
'0x1L'
```

如果x不是Python的int或者long对象，它必须定义**index**()方法以返回一个整数。

参见int()，它将十六进制字符串转化成一个整数。

注意

使用float.hex()方法得到浮点数的十六进制字符串表示。

改变于版本2.4：在此之前只返回无符号数。

id(*object*) 返回对象的“标识”。这是一个整数（或长整数），保证在对象的生命期内唯一且不变。生命期不重叠的两个对象可以有相同的id()值。

CPython实现细节：这是对象的内存地址。

input([*prompt*]) 等同于eval(raw_input(prompt))。

该函数不会捕获用户错误。如果输入语法不合法，将抛出SyntaxError。如果演算中有错误，将抛出其它异常。

如果有装载readline，input()将会用它来提供复杂的行编辑和历史功能。

考虑使用raw_input()函数来得到用户的一般输入。

int(*x=0*)int(*x*, *base=10*) 将数字或字符串*x*转化成一个整数，如果没有参数则返回0。如果*x*是个数字，它可以是原始／长整数，或者浮点数。如果*x*是浮点数，则向0截断。如果参数超出了整数的范围，则返回长整数对象。

如果*x*不是个数字，或者存在*base*参数，则*x*必须是个表示以*base*为基数的*integer literal*（整数字面量）的字符串或者Unicode对象。字面量的前面可以有+或者-（中间不能有空格），周围可以有空白。以n为基数的字面量包含数字0到n-1，用a到z（或者A到Z）来表示10到35。默认的*base*是10。允许的值为0和2-36。二进制，八进制和十六进制的字面量前面可以有0b/0B，0o/0O/0，或者0x/0X，就像代码中的整数字面量一样。基数0表示严格按整数字面量来解释字符串，所以实际的基数为2，8，10或者16。

整数类型描述于*Numeric Types — int, float, long, complex*。

isinstance(*object*, *classinfo*) 如果参数*object*是参数*classinfo*的一个实例；或者是一个子类（直接的，间接的，或者*virtual*），返回真。如果*classinfo*是类型对象（新式类）而*object*是该类型对象；或者是其子类（直接的，间接的，或者*virtual*），返回真。如果*object*不是给定

类型的类实例或者对象，该函数总是返回假。如果*classinfo*既不是类对象，也不是类型对象，它可以是类／类型对象的元组，或者递归包含这样的元组（不接受其它的序列类型）。如果*classinfo*不是类，类型，类／类型的元组，抛出TypeError异常。

改变于版本2.2：添加对类型信息的元组的支持。

issubclass(*class*, *classinfo*) 如果*class*是*classinfo*的子类（直接的，间接的，或者*virtual*），返回真。一个类被认为是它自己的子类。*classinfo*可以是类对象的元组，这时*classinfo*中的每个类对象都会被检查。其它情况下，抛出TypeError异常。

改变于版本2.3：添加对类型信息的元组的支持。

iter(*o*[, *sentinel*]) 返回一个*iterator*对象。根据有无第二个参数，对第一个参数的解释相差很大。如果没有第二个参数，*o*必须是个集合对象，要么支持迭代协议（即 **iter()**方法），要么支持序列协议（即**getitem()**方法，整数参数从0开始）。如果这些协议都不支持，抛出TypeError。如果有第二个参数*sentinel*，*o*必须是个可调用对象。这种情况下返回的迭代，每当调用其next()方法时，将会调用*o*（不带参数）；如果返回值等于*sentinel*，抛出StopIteration，否则返回该值。

第二种形式的iter()的一个有用的应用就是读一个文件的行，直到读到特定行。下面的例子读一个文件，直到readline()方法返回一个空字符串：

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

出现于版本2.2。

len(*s*) 返回对象的长度（元素的个数）。参数可以是序列（如字符串，字节，元组，列表或者范围）或者集合（如字典，集合或者固定集合）。

list([*iterable*]) 返回一个列表，其元素来自于*iterable*（保持相同的值和顺序）。*iterable*可以是个序列，支持迭代的容器，或者迭代器对象。如果*iterable*已经是个列表，返回其拷贝，类似于iterable[:]。例如，list('abc')返回['a','b','c']，list((1,2,3))返回[1,2,3]。如果没有参数，返回一个新的空的列表，[]。

list是可变序列类型，见文档*Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange*。关于其它容器参见内置dict，set，和tuple类，以及collections模块。

locals() 更新并返回表示当前局部符号表的字典。当locals在函数块中而不是类块中被调用时，locals()返回自由变量。

注意

不应该修改该字典的内容；所做的改变不一定会影响到解释器所用的局部和自由变量的值。

long(*x=0*)long(*x*, *base=10*) 将一个字符串或者数字转化成一个长整数。如果参数时个字符串，它必须包含一个数字，任意大小，可以有符号，周围可以有空白。*base*参数和int()中的一样，只有当*x*是字符串的时候才能有此参数。其它情况下，参数可以是个原始／长整数，或者浮点数，返回具有相同值的长整数。浮点数转成整数时将浮点数向零截断。如果没有参数，返回0L。

长整数类型描述于*Numeric Types — int, float, long, complex*。

map(*function*, *iterable*, *...*) 将*function*应用于*iterable*的每一个元素，返回结果的列表。如果有额外的*iterable*参数，并行的从这些参数中取元素，并调用*function*。如果一个参数比另外的要短，将以None扩展该参数元素。如果*function*是None使用特性函数；如果有多个参数，map()返回一元组列表，元组包含从各个参数中取得的对应的元素（某种变换操作）。*iterable*参数可以是序列或者任意可迭代对象；结果总是列表。

max(*iterable*[, *key*])max(*arg1*, *arg2*, **args*[, *key*]) 返回可迭代的对象中的最大的元素，或者返回2个或多个参数中的最大的参数。

如果有一个位置参数，*iterable*必须是个非空的可迭代对象（如非空字符串，元组或者列表）。返回可迭代对象中最大的元素。如果有2个或更多的位置参数，返回最大位置参数。

可选的*key*参数指明了有一个参数的排序函数，如list.sort()中使用的排序函数。如果有*key*参数，它必须是关键字参数（例如，max(a,b,c,key=func)）。

改变于版本2.5：添加了对可选参数*key*的支持。

memoryview(*obj*) 返回给定参数的“内存视图”。参见*memoryview type*。

min(*iterable*[, *key*])min(*arg1*, *arg2*, **args*[, *key*]) 返回可迭代的对象中的最小的元素，或者返回2个或多个参数中的最小的参数。

如果有一个位置参数，*iterable*必须是个非空的可迭代对象（如非空字符串，元组或者列表）。返回可迭代对象中最小的元素。如果有2个或更多的位置参数，返回最小的位置参数。

可选的*key*参数指明了有一个参数的排序函数，如list.sort()中使用的排序函数。如果有*key*参数，它必须是关键字参数（例如，min(a,b,c,key=func)）。

改变于版本2.5：添加了对可选参数*key*的支持。

next(*iterator*[, *default*]) 通过调用*iterator*的next()方法，得到它的下一个元素。如果有*default*参数，在迭代器迭代完所有元素之后返回该参数；否则抛出StopIteration。

出现于版本2.6。

object() 返回一个新的无特征的对象。object是所有新式类的基类。它有对所有新式类的实例通用的方法。

出现于版本2.2。

改变于版本2.3：改函数不接受任何的参数。在以前，它接受参数，但是会被忽略掉。

oct(*x*) 将一个（任意尺寸）整数转化成一个八进制字符串。结果是一个合法的Python表达式。

改变于版本2.4：以前只返回一个无符号数字面量。

open(*name*[, *mode*[, *buffering*]]) 打开一个文件，返回一个file类型的对象，file类型描述于*File Objects*章节。如果文件不能打开，抛出IOError。当要打开一个文件，优先使用open()，而不是直接调用file构造函数。

头两个参数类似于stdio‘s fopen()的参数：*name*是要打开的文件的名字，*mode*是个指示如何打开文件的字符串。

*mode*的常用值包括：'r'读文件；'w'写文件（如果文件存在则截断之）；'a' 附加（在 某些Unix系统上意味着所有的写操作附加到文件的末尾，不管当前的写位置）。如果没有*mode*，默认是'r'。默认使用文本模式，它会在写文件时将'\n'字符转化成平台特定的字符，在读文件时又转回来。因此在打开二进制文件的时候，要以二进制模式打开文件，把'b'添加到*mode*值，这样可以增强可移植性。（在不区分二进制文件和文本文件的系统上，附加'b'仍然时有用的，它可以起到文档的目的。）参见下文以得到*mode*更多的可能的值。

可选的*buffering*参数指明了文件需要的缓冲大小：0意味着无缓冲；1意味着行缓冲；其它正值表示使用参数大小的缓冲（大概值，以字节为单位）。负的*buffering*意味着使用系统的默认值，一般来说，对于tty设备，它是行缓冲；对于其它文件，它是全缓冲。如果没有改参数，使用系统的默认值。[2]

模式'r+'，'w+'和'a+'打开文件以便更新（同时读写）。注意'w+'会截断文件。在区分文本文件和二进制文件的系统上，在模式中附加'b'会以二进制模式打开文件；在不区分的系统上，添加'b'没有效果。

除了标准的fopen()模式值，*mode*可以是'U'或者'rU'。构建Python时一般会添加*universal newlines*（统一新行）支持；'U'会以文本模式打开文件，但是行可以以以下字符结束：'\n'（Unix行结束符），'\r'（Macintosh惯例），或者'\r\n'（Windows惯例）。Python程序会认为它们都是'\n'。如果构建Python时没有添加统一新行的支持，*mode*'U'和普通文本模式一样。注意，这样打开的文件对象有一个叫newlines的属性，它的值是None（没有发现新行），'\n'，'\r'，'\r\n'，或者是包含所有已发现的新行字符的元组。

Python要求模式字符串在去除'U'后以'r'，'w'或者'a'开头。

Python提供了许多文件处理模块，包括fileinput，os，os.path，tempfile和shutil。

改变于版本2.5：引入了对模式字符串第一个字符的限制。

ord(*c*) 给定一个长度为一的字符串，如果参数是unicode对象，则返回表示字符的代码点的整数；如果参数是八位字符串，返回字节值。例如，ord('a')返回整数97，ord(u'\u2020')返回8224。它是八位字符串chr()的反函数，也是unicode对象unichr()的反函数。如果参数是

unicode且构建Python时添加了UCS2 Unicode支持，那么字符的码点必须在[0..65535]的闭区间；如果字符串的长度为2，则抛出TypeError。

pow(*x*, *y*[, *z*]) 返回*x* 的 *y*次幂;如果 *z* 提供的时候，, 返回 *x* 的 *y* 次幂，然后对 *z* 取模。(这样比pow(x,y)%z) 更高效。两个参数的形式 pow(x,y) 与使用 操作符: x**y 是等价的。

参数必须是数字类型的。由于操作数是混合类型的，二进制计算的原因需要一些强制的规定。对于整型和长整型的操作数，计算结果和操作数（强制后的）是相同的类型。除非第二个 参数是负数。在这种情况下, 所有的参数都会被转化成浮点型，并且会返回一个浮点的结果。例如, 10**2** 返回 **100,** 但 **10**-2 返回0.01.(这个新特性被加入在Python2.2中在Python2.1和之前的版本中,如果两个参数是整型，并且第二个参数为负数的情况下，会抛出一个异常。）如果第二个参数为负数，那么第三个参数必须省略。如果提供参数 *z* , *x* and *y* 必须为整数，而且*y*要是非负整数。(这个限制是在Python2.2加入的。Python2.1以及之前的版本, 三个参数都是浮点型的pow() 版本，返回的结果依赖 平台 对于浮点数的取整情况。）

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*) 以sep分割，end的值结尾，将 目标对象 打印到 文件流中。*sep*, *end* 和 *file*，如果提供这三个参数的话，必须以键值的形式。

所有非键值形式提供的参数，都被转化为字符串，就像用str()转化那样。然后写到 文件流中，以 *sep* 分割， *end* 结尾。*sep* and *end* 都必须是字符串形式的;也可以留成 None, 这样会使用默认值。如果没有打印 对象, print() 只打印一个 结束符号 *end*.

file 参数一定要是含有 write(string)方法的对象 ;如果该参数为空，或为None, 默认使用sys.stdout 作为输出。输出缓冲方式由*file*决定。例如， 使用file.flush() 来确保, 立即显示在屏幕上.

Note

This function is not normally available as a built-in since the name print is recognized as the print statement.为了使得print语句失效，而使用 print() 函数, 可以使用future 语句 在你的模块上面:

```
from __future__ import print_function
```

2.6 版中新增。

property([*fget*[, *fset*[, *fdel*[, *doc*]]]]) 返回新式类（继承自object的类）的一个属性。

*fget*是用于获取属性值的函数， 类似地*fset*用于设置属性，*fdel*用于删除属性。典型的用法是定义一个托管的属性：x：

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

假设*c*是*C*的一个实例，c.x将调用获取函数，c.x=value调用设置函数，delc.x调用删除函数。

如果给出*doc*，它将是该属性的文档字符串。否则，该属性将拷贝*fget*的文档字符串（如果存在）。这使得用property()作为装饰器创建一个只读属性非常容易：

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

将voltage()方法转换为一个与只读属性具有相同名称的获取函数。

A property object has getter, setter, and deleter methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. 最好的解释就是使用一个例子：

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

这段代码与第一个例子完全相等。请确保给额外的函数与原始的属性相同的名字（此例中为x。）

The returned property also has the attributes fget, fset, and fdel corresponding to the constructor arguments.

2.2版中新增。

2.5版中的变化：如果*doc*没有给出则使用*fget*的文档字符串。

2.6版中的变化：添加getter、setter和deleter属性。

range(*stop*)range(*start*, *stop*[, *step*]) 这是一个创建算术级数列表的通用函数。它最常用于for循环。参数必须为普通的整数。如果*step*参数省略，则默认为1。如果*start*参数省略，则默认为0。该函数的完整形式返回一个整数列表[start,start+step,start+2*step,...]*。如果step为正，则最后一个元素*start*+*istep最大且小于*stop*；如果*step*为负，则最后一个元素start+i*step*最小且大于stop。step*必须不能为零（否则会引发ValueError）。示例：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([*prompt*]) 如果有*prompt*参数，则将它输出到标准输出且不带换行。该函数然后从标准输入读取一行，将它转换成一个字符串（去掉一个末尾的换行符），然后返回它。当读到EOF时，则引发EOFError。示例：

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

如果已经加载readline模块，那么raw_input()将用它来提供优雅的行编辑和历史功能。

reduce(*function*, *iterable*[, *initializer*]) 将带有两个参数的*function*累计地应用到*iterable*的元素上，从左向右，以致将可迭代序列reduce为一个单一的值。例如，reduce(lambdax,y:x+y,[1,2,3,4,5])相当于计算((((1+2)+3)+4)+5)。左边的参数*x*是累计之后的值，右边的参数*y*是来自

*iterable*中的修改值。如果提供可选的参数*initializer*，它在计算时放在可迭代序列的最前面，并且当可迭代序列为空时作为默认值。如果*initializer*没有给出，且*iterable*只包含一个元素，将返回第一个元素。大致等同于：

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

reload(*module*) 重载之前已经引入过的模块.参数必须是一个模型对象，所以之前它必须成功导入。如果你用外部编辑器编辑了模型的源文件并且打算在不离开python解释器的情况下使用模型的新版本，reload将非常有用。返回值是该模块对象（与*module*参数相同）。

当执行reload(module)时：

- python模型的代码将重新编译并且模型级的代码会重新执行，定义一系列对象，这些对象的名字和模型字典中的名字相关联。改进后的模型的 init 函数不会第二次加载。
- 跟Python其他对象一样，旧的对象只有在他们的引用计数将为0的时候被系统收回。
- 模型命名空间中的名字自动升级指向新的或者修改后的对象。
- 对旧对象的其他引用（比如模型延展的命名）不会连接到新的改进对象。每一个命名空间在被请求的时候，都必须更新。

这里有一些列的其他警告：

If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in sys.modules.To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can reload() it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained.Redefinitions of names will override the old definitions, so this is generally not a problem.If the new version of a module does not define a name that was defined by the old version, the old definition remains.This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a try statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for sys, **main** and **builtin**.In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using from ...import ..., calling reload() for the other module does not redefine the objects imported from it — one way around this is to re-execute the from statement, another is to use import and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition.The same is true for derived classes.

repr(*object*) 返回某个对象可打印形式的字符串。它与字符串转换式（反引号）产生的值相同。有时候能够把这个操作作为一个普通的函数访问非常有用。For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object.类可以通过定义**repr**()方法控制该函数对其实例的返回。

reversed(*seq*) 返回一个反转的迭代器。*seq*必须是一个具有**reversed**() 方法或支持序列协议的对象（整数参数从0开始的**len**()方法和**getitem**() 方法）。

2.4版中新增。

2.6版中的变化：添加可以编写一个定制的**reversed**()方法的可能。

round(*number*[, *ndigits*]) 返回一个浮点型 近似值，保留小数点后 *ndigits* 位。如果省略*ndigits*，它默认为零。结果是一个浮点数。Values are rounded to the closest multiple of 10 to the power minus *ndigits*;if two multiples are equally close, rounding is done away from 0（所以，例如，round(0.5)是1.0 且round(-0.5)是-1.0）。

Note

浮点数round()的行为可能让人惊讶，例如round(2.675,2)给出的是2.67 而不是期望的2.68。这不是一个错误：大部分十进制小数不能用浮点数精确表示，它是因为这样的一个事实的结果。更多信息，请参阅*Floating Point Arithmetic: Issues and Limitations*。

set([*iterable*]) 返回一个新的set 对象，其元素可以从可选的*iterable*获得。set是一个内建的类。关于该类的文档，请参阅set和集合类型 — *set, frozenset*。

关于其它容器请参阅内建的frozenset、list、tuple和dict类，还有collections模块。

2.4版中新增。

setattr(*object*, *name*, *value*) getattr()的相反操作。参数是一个对象、一个字符串和任何一个值。字符串可以是一个已存在属性的名字也可以是一个新属性的名字。该函数将值赋值给属性，只要对象允许。例如，setattr(x,'foobar',123)等同于x.foobar=123。

slice(*stop*)slice(*start*, *stop*[, *step*]) 返回一个*slice*对象，表示由索引range(start,stop,step)指出的集合。*start*和*step*参数默认为None。切片对象具有只读属性start、stop和step，它们仅仅返回参数的值（或者它们的默认值）。它们没有其他显式的函数；它是它们用于Numerical Python和其它第三方扩展。在使用扩展的索引语法时同样会生成切片对象。例如：a[start:stop:step]或a[start:stop,i]。返回迭代器的另外一个版本可以参阅itertools.islice()。

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]]) 依据*iterable*中的元素返回一个新的列表。

可选参数*cmp*、*key*和*reverse*与list.sort()方法的参数含义相同（在可变的序列类型一节描述）。

*cmp*指定一个定制化的带有两个参数的比较函数（可迭代的元素），它应该根据第一个参数是小于、等于还是大于第二个参数返回负数、零或者正数：cmp=lambdax,y:cmp(x.lower(),y.lower())。默认值是None。

*key*指定一个带有一个参数的函数，它用于从每个列表元素选择一个比较的关键字：key=str.lower。默认值是None（直接比较元素）。

*reverse*是一个布尔值。如果设置为True，那么列表元素以反向比较排序。

通常情况下，*key*和*reverse*转换处理比指定一个等同的*cmp*函数要快得多。这是因为*cmp*为每个元素调用多次但是*key*和*reverse*只会触摸每个元素一次。使用functools.cmp_to_key()来转换旧式的*cmp*函数为*key*函数。

关于排序的实例和排序的简明教程，请参阅Sorting HowTo。

2.4版中新增。

staticmethod(*function*) 返回*function*的一个静态方法。

静态方法不接受隐式的第一个参数。要声明静态方法，请使用下面的习惯方式：

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
        ...
```

@staticmethod形式是一个函数装饰器 – 细节请参阅函数定义中函数定义的描述。

它既可以在类上调用（例如C.f()）也可以在实例上调用（例如C().f()）。除了它的类型，实例其他的内容都被忽略。

Python中的静态方法类似于Java或C++。关于创建类构造器的另外一种方法，请参阅classmethod()。

更多关于静态方法的信息，请查看标准类型层次中标准类型层次的文档。

2.2版中新增。

2.4版中的新变化：添加函数装饰器语法。

str(*object=''*) 返回一个字符串，包含对象的友好的可打印表示形式。对于字符串，它返回字符串本身。与repr(object)的区别是str(object)不会永远试图返回一个eval()可接受的字符串；它的目标是返回一个可打印的字符串。如果没有给出参数，则返回空字符串''。

关于字符串更多的信息请参阅序列类型 — *str, unicode, list, tuple, bytearray, buffer, xrange*，它描述序列的函数（字符串是序列的一种），以及在*String Methods* 一节中描述的字符串自己的方法。若要输出格式化的字符串，请使用模板字符串或在字符串格式化操作一节中描述的%操作符。另外可参阅字符串服务一节。另请参阅unicode()。

sum(*iterable*[, *start*]) 将*start*以及*iterable*的元素从左向右相加并返回总和。*start*默认为0。*iterable*的元素通常是数字，start值不允许是一个字符串。

对于某些使用场景，有比sum()更好的选择。连接字符串序列的首选和快速的方式是调用''.join(sequence)。如要相加扩展精度的浮点数，请参阅math.fsum()。若要连接一系列的可迭代量，可以考虑使用itertools.chain()。

2.3版中新增。

super(*type*[, *object-or-type*]) 返回一个代理对象，这个对象指派方法给一个父类或者同类.这对进入类中被覆盖的继承方法非常有用。搜索顺序和 getattr() 一样。而它自己的 类型 则被忽略。

类型的 **mro** 方法 *type* 罗列了被getattr() 和 super(). 用来搜索顺序的解决方法。The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound.If the second argument is an object, isinstance(obj,type) must be true.If the second argument is a type, issubclass(type2,type) must be true (this is useful for classmethods).

Note

super() only works for *new-style class*es.

There are two typical use cases for *super*.In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable.This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment.This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance.This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method.Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

Note that super() is implemented as part of the binding process for explicit dotted attribute lookups such as super().**getitem**(name).It does so by implementing its own **getattribute**() method for searching classes in a predictable order that supports cooperative multiple inheritance.Accordingly, super() is undefined for implicit lookups using statements or operators such as super()[name].

Also note that super() is not limited to use inside methods.The two argument form specifies the arguments exactly and makes the appropriate references.

For practical suggestions on how to design cooperative classes using super(), see guide to using super().

2.2版中新增。

tuple([*iterable*]) 返回一个元组，其元素及顺序与*iterable*的元素相同。*iterable*可以是一个序列、支持迭代操作的容器或迭代器对象。如果*iterable*已经是一个元组，它将被原样返回。例如，tuple('abc')返回('a','b','c')，tuple([1,2,3])返回(1,2,3)。如果没有给出参数，则返回一个空的元组()。

tuple是一个不可变序列类型，其文档在序列类型 — *str, unicode, list, tuple, bytearray, buffer, xrange*。关于其它容器，请参阅内建的dict、list和set类以及collections模块。

type(*object*)type(*name*, *bases*, *dict*) 只有一个参数时，返回*object*的类型。返回值是一个类型对象。建议使用内建函数isinstance()测试一个对象的类型。

带有三个参数时，返回一个新的类型对象。它本质上是class语句的动态形式。*name*字符串是类的名字且将成为**name**属性；*bases*元组逐条列举基类并成为**bases**属性；*dict*字典是包含类体定义的命名空间并成为**dict**属性。例如，下面的两条语句创建完全相同的type对象：

```
>>> class X(object):
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

版本2.2中新增。

unichr(*i*) 返回Unicode码为整数*i*的Unicode字符。例如，unichr(97)返回字符串u'a'。这是Unicode字符串ord()的反转。其参数合法的范围取决于Python是如何配置的 – 它可以是UCS2 [0..0xFFFF] 或者UCS4 [0..0x10FFFF]。否则引发ValueError。对于ASCII和8比特字符串，请参阅chr()。

2.0版中新增。

unicode(*object=''*)unicode(*object*[, *encoding*[, *errors*]]) 使用下面的一种模式返回*object*的Unicode版字符串：

If *encoding* and/or *errors* are given, unicode() will decode the object which can either be an 8-bit string or a character buffer using the codec for *encoding*.The *encoding* parameter is a string giving the name of an encoding;if the encoding is not known, LookupError is raised.Error handling is done according to *errors*;this specifies the treatment of characters which are invalid in the input encoding.If *errors* is 'strict' (the default), a ValueError is raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, U+FFFD, to be used to replace input characters which cannot be decoded.See also the codecs module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings.More precisely, if *object* is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a **unicode**() method, it will call this method without arguments to create a Unicode string.For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in 'strict' mode.

For more information on Unicode strings see *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange* which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the *String Methods* section.To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section.In addition see the *String Services* section.See also str().

2.0版中新增。

2.2版中的变化：Support for unicode() added.

vars([*object*]) 返回模块、类、实例或其它任何具有**dict**属性的对象的**dict**属性。

模块和实例这样的对象具有可更新的**dict**属性；然而，其它对象可能对它们的**dict**属性具有写限制（例如，新式类使用dictproxy 来防止直接的字典更新）。

如果不带参数，则vars()的行为类似locals()。注意，局部字典只用于读取因为对局部字典的更新被忽略。

xrange(*stop*)xrange(*start*, *stop*[, *step*]) 该函数与range()非常相似，但是它返回一个*xrange 对象*而不是一个列表。这是一个惰性的序列类型，它生成与对应的列表相同的值但不会真正同时一起存储它们。xrange()相比range()的优点不大（因为xrange()仍然必须创建需要的值），除非在内存紧张的机器上使用一个非常大的range或者range的所有元素从不会使用（例如当循环经常被break终止）。关于xrange对象的更多对象，请参阅*XRange 类型*和*序列类型—str, unicode, list, tuple, bytearray, buffer, xrange*。

CPython实现细节：xrange()的设计意图是简单而快速。具体的实现可能强加各种限制以实现这点。Python的C实现限制所有的参数为C的原生长整型（Python的“短”整数），且要求元素的个数适合一个C的原生长整型。如果需要一个更大的range，可以使用itertools模块创建一个另外的版本：islice(count(start,step),(stop-start+step-1+2*(step<0))//step)。

zip([*iterable*, ...]) 该函数返回一个元组的列表，其中第*i*个元组包含每个参数序列的第*i*个元素。返回的列表长度被截断为最短的参数序列的长度。当多个参数都具有相同的长度时，zip()类似于带有一个初始参数为None的map()。只有一个序列参数时，它返回一个1元组的列表。没有参数时，它返回一个空的列表。

可以保证迭代按从左向右的计算顺序。这使得使用zip(*[iter(s)]*n)来将一系列数据分类归并为长度为n的组成为习惯用法。

zip()与 * 操作符一起可以用来unzip 一个列表：

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

版本2.0中新增。

2.4版中的变化：以前，zip()要求至少一个参数，且zip()将引发TypeError而不是一个空序列。

import(*name*[, *globals*[, *locals*[, *fromlist*[, *level*]]]])

注意

与importlib.import_module()不同，这是一个高级的函数，不会在日常的Python变成中用到，。

This function is invoked by the import statement.It can be replaced (by importing the **builtin** module and assigning to **builtin.import**) in order to change semantics of the import statement, but nowadays it is usually simpler to use import hooks (see **PEP 302**).Direct use of **import**() is rare, except in cases where you want to import a module whose name is only known at runtime.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context.The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*.The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the import statement.

level specifies whether to use absolute or relative imports.The default is -1 which indicates both absolute and relative imports will be attempted.0 means only perform absolute imports.Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling **import**().

When the *name* variable is of the form package.module, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*.However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement importspam results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], -1)
```

The statement importspam.ham results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], -1)
```

Note how **import**() returns the toplevel module here because this is the object that is bound to a name by the import statement.

On the other hand, the statement fromspam.hamimporteggs,sausageassaus results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the spam.ham module is returned from **import**().From this object, the names to import are retrieved and assigned to their respective names.

如果你只是简单地想依据名字导入一个模块，请使用importlib.import_module()。

2.5版中的变化：添加level参数。

2.5版中的变化：添加关键字参数的支持。

3. 不太重要的内建函数

有几个内建的函数在现代Python编程中已经没有必要再学习、知道和使用了。这里保留它们是为了保持为老版本Python编写的程序的向后兼容。

Python程序员、培训人员、学生以及书籍编写人员应该自由跳过这些函数而不用担心遗漏重要的内容。

apply(*function*, *args*[, *keywords*]) The *function* argument must be a callable object (a user-defined or built-in function or method, or a class object) and the *args* argument must be a sequence.The *function* is called with *args* as the argument list;the number of arguments is the length of the tuple.If the optional *keywords* argument is present, it must be a dictionary whose keys are strings.It specifies keyword arguments to be added to the end of the argument list.Calling apply() is different from just calling function(args), since in that case there is always exactly one argument.The use of apply() is equivalent to function(*args,* *keywords).

自2.3版后废弃：使用function(*args,* ***keywords)*来代替*apply(function,args,keywords)*（参见*[*Unpacking Argument Lists*](#)）。

buffer(*object*[, *offset*[, *size*]]) *object*参数必须是一个支持缓冲区调用接口的对象（例如字符串、数组和缓冲区）。将创建一个新的引用*object*参数的缓冲区对象。缓冲区对象将是一个从*object*的起点开始的切片（或者从指定的*offset*）。切片将扩展到*object*对象的末尾（或者通过*size*参数指定的长度）。

coerce(*x*, *y*) Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations.If coercion is not possible, raise TypeError.

intern(*string*) Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy.Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare.Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

2.3版中的变化：Interned strings are not immortal (like they used to be in Python 2.2 and before);you must keep a reference to the return value of intern() around to benefit from it.

脚注

[1]	它极少被使用，所以不应该成为语句。

[2]	当前在没有**setvbuf()**的系统上指明缓冲大小是没有效果的。该接口指明缓冲大小不是通过调用**setvbuf()**函数来实现的，因为在执行任意的**I/O**操作后再调用该函数可能会导致内存转储，同时也没有一个可靠的方法来判断这种情况。

[3]	In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (such as modules) can be.This may change.

4. 内建的常量

内置的命名空间中存在少数几个常量。它们是：

False bool类型值 false。

版本 2.3中新增。

True bool类型值true。

版本 2.3中新增。

None types.NoneType的唯一值。None常用来表示缺少的值，例如当默认参数没有传递给函数时。

2.4 版本中的更改：对None赋值变成非法且引发SyntaxError。

NotImplemented 它可以由特殊的"rich comparison"方法（**eq()**，**lt()**以及类似的方法）返回，以指示另一种类型没有实现这种比较操作。

Ellipsis 与扩展的切片语法一起使用的特殊值。

debug 如果Python没有以*-O*选项启动则该常量为真。另请参阅assert语句。

注

None和**debug**的名称不能重新赋值（如果对它们赋值，即使作为一个属性名称，也会引发SyntaxError），所以它们可以被认为是"真实"的常量。

2.7 版本中的更改：将**debug**作为一个属性来赋值变成非法。

4.1. site模块添加的常数

site模块（在启动期间自动导入，除非给出*-S*命令行选项）将添加几个常数到内置的命名空间。它们可用于交互式解释器的shell，不应在程序中使用。

quit([*code=None*])exit([*code=None*]) 当打印这两个对象时打印一条类似（"Use quit() or Ctrl-D (i.e. EOF) to exit"）的信息，当它们被调用时则使用指定的退出码引发SystemExit 。

copyrightlicensecredits 当打印这两个对象时打印一条类似（"Type license() to see the full license text"）的信息，当它们被调用时则以分页显示相应的文本。

5. 内建的类型

以下各节描述内置于解释器的标准类型。

注

历史上（直到2.2版的发布），Python 的内置类型不同于用户定义的类型，因为不可能用内置类型作为面向对象继承的基类。这种限制不再存在。

主要的内置类型为数字、 序列、 映射、 文件、 类、 实例和异常。

某些操作被几种对象类型支持；特别需要注意的是，几乎所有对象都可以比较、测试真值、转换为字符串（其实就是用*repr()*函数，或略有差异的str()函数来转换）。后者在对象使用print()函数写出时隐式地调用。

5.1. 真值的测试

任何对象都可以测试真值，用于if或while的条件或下面布尔运算的操作数。下面的值被视为假：

- None
- False
- 任何数值类型的零，例如，0、 0 L、 0.0、 0j。
- 任何空的序列，例如， ''、 ()、 []。
- 任何空的映射，例如，{}。
- 用户定义的类的实例，如果该类定义一个**nonzero()**或**len()**的方法，在该方法返回整数零或布尔值False时。[1]

所有其他值都被视为真 — 所以许多类型的对象永远为真。

结果为布尔值的运算和内建函数总是返回0或False表示假以及1或True表示真，除非另有说明。（重要的例外：布尔操作符or和and始终返回它们的一个操作数。）

5.2. 布尔操作 — **and**, **or**, **not**

这些是布尔操作，按升序优先排序：

操作	结果	注
xory	如果*x*为假，那么返回*y*，否则返回*x*	(1)
xandy	如果*x*为假，那么返回*x*，否则返回*y*	(2)
notx	如果*x*为假，那么返回True，否则返回False	(3)

注：

1. 这是一个短路操作符，因此只有第一个参数为False时才计算第二个参数。
2. 这是一个短路操作符，因此只有第一个参数为True时才计算第二个参数。
3. not比非布尔操作符的优先级低，因此nota==b解释为not(a==b)，a==notb是一个语法错误。

5.3. 比较操作

所有对象都支持比较操作。它们都具有相同的优先级（高于布尔操作）。比较可以任意链接；例如，x<y< =z相当于x<yandy< =z，只是*y*只计算一次（但这两种情况在x<y是假时都不会计算*z*）。

下表汇总了比较操作：

操作	含义	注
<	严格地小于	
<=	小于或等于	
>	严格地大于	
>=	大于或等于	
==	等于	
!=	不等于	(1)
is	对象的ID	
isnot	不同的对象ID	

注：

1. !=也可以写成<>，但这只是用于保持向后兼容性的用法。新的代码应该一直使用!=。

不同类型的对象，不同的数值和字符串类型除外，比较结果永远不会相等；这类对象排序的结果永远一致但是顺序却是随机的（使得异构数组的排序可以生成一致的结果）。此外，某些类型（例如，文件对象）只支持退化的比较概念，该类型的任何两个对象都不相等。同样，这类对象排序的顺序是随机的但是会永远是一致的。当任何一个操作数是复数时，<、=、 >和> =运算符会引发TypeError异常。

类的非同一个实例比较时通常不相等，除非该类定义eq()或cmp()方法。

一个类的实例通常不能与同一个类的其它实例或者其他类型的对象排序，除非该类定义足够丰富的比较方法（ge()、le()、gt()、lt()）或cmp()方法。

CPython 的实现细节：除数值以外不同类型的对象按它们的类型名称进行排序；不支持合适比较的相同类型的对象按它们的地址进行排序。

还有两个具有相同优先级的操作in和notin只支持序列类型 (见下文)。

5.4. 数值类型 — int, float, long, complex

有四种不同的数值类型：普通整数、长整数、浮点数和复数。此外，布尔值是普通整数的一个子类型。普通整数（也被只叫做整数）使用C中的long实现，其精度至少为32位（sys.maxint始终设置为当前平台最大的普通整数值，最小值是-sys.maxint-1)。长整数具有无限的精度。浮点数字通常使用C中的double实现；有关你的程序所运行的机器上的浮点数精度及其内部表示形式的信息在sys.float_info中可以获得。复数有实部和虚部，各是一个浮点数。若要从复数z中提取这些部分，请使用z.real和z.imag。（标准库包括额外的数值类型，fractions支持有理数，decimal支持用户自定义精度的浮点数。）

数值通过数字字面值或内建的函数和操作的结果创建。普通的整数字面值（包括二进制、十六进制和八进制数字）产生普通整数，除非它们指定的值太大以致不能用一个普通的整数表示，在这种情况下它们产生一个长整型。带有'L'或'l'后缀的整数字面值产生长整数（偏向使用'L'，因为1l看起来太像十一）。包含小数点或指数符号的数字字面值产生浮点数。将'j'或'J'附加到数字字面值的尾部产生实部为零的复数。复数字面值是实部和虚部的和。

Python完全支持混合的算法：当二元算术运算符的操作数是不同的数值类型时，“较窄”类型的操作数会拓宽成另外一个操作数的类型，其中整数窄于长整数窄于浮点数窄于复数。比较混合型数字之间使用相同的规则。[2]构造函数int()、long()、float()和complex()可用于产生的一种特定类型的数值。

所有内置的数值类型都支持以下操作。运算符的优先级请参阅幂运算符和后面几节。

操作	结果	注
x+y	*x*与*y*和	
x-y	*x*与*y*的差	
x*y	*x*与*y*的积	
x/y	*x*与*y*的商	(1)
x//y	*x*与*y*的（整除）商	(4)(5)
x%y	x/y的余数	(4)
-x	负 *x*	
+x	*x*保持不变	
abs(x)	*x*的绝对值或大小	(3)
int(x)	*x*转换成整数	(2)
long(x)	*x*转换成长整数	(2)
float(x)	*x*转换成浮点数	(6)
complex(re,im)	实部为*re*，虚部为*im*的一个复数。*im*默认为零。	
c.conjugate()	复数*c*的共轭。（用实数表示）	
divmod(x,y)	元组(x//y,x%y)	(3)(4)
pow(x,y)	*x*的*y*次方	(3)(7)
x**y	*x*的*y*次方	(7)

注：

1. 对于（普通或长）整数除法，结果是一个整数。结果总是向负无穷舍入：1/2是0，(-1)/2是-1，1/(-2)是-1，（-1)/(-2)是0。请注意如果任何一个操作数是长整数，结果都会是一个长整数，与值大小无关。
2. 使用int()或long()函数转换浮点数会向零截断，类似相关的函数math.trunc()函数。使用函数math.floor()以向下取整和math.ceil()以向上取整。
3. 完整的说明请参阅内置函数。
4. 从2.3版开始弃用：整除运算符、取模运算符和divmod()函数不再为复数定义。相反，如果合适，可以使用abs()函数转换为浮点数。
5. 也被称为整数除法。结果的值完全是一个整数，虽然结果的类型不一定是整型。
6. 浮点数还接受可带有可选前缀 "+"或"-"的字符串"nan"和"inf"来表示非数字（NaN））和正/负无穷。

在2.6版中新增。

1. Python定义pow(0,0)和0**0为1，这对于编程语言很常见。

所有的numbers.Real类型（int、long和float）还包含以下的操作：

操作	结果	注
math.trunc(x)	*x*截取成整数	
round(x[,n])	*x*舍入到n位，舍入ties away from零。如果n省略，默认为0。	
math.floor(x)	<= *x*的最大浮点整数	
math.ceil(x)	>= *x*的最小浮点整数	

6. 内建的异常

异常应该是类对象。异常定义在模块exceptions中。该模块不需要显式导入：这些异常在内置命名空间中有提供，就和exceptions模块一样。

对于类异常，如果在try语句的except子句中提到一个类，该子句还会处理任何从那个类派生的异常类（不是它派生自的异常类）。通过子类化得到的两个不相关的异常类永远不会相等，即使它们具有相同的名称。

下面列出的异常可以通过解释器或内置函数生成。除了提到的那些地方，它们还有"相关联的值"指示错误的详细的原因。它可能是一个字符串或一个包含几项信息（例如，错误码和解释代码的字符串）的元组。关联的值为raise语句的第二个参数。如果异常类派生自标准的根类BaseException，关联的值作为异常实例的args属性呈现。

用户代码可以引发内置异常。这可以用来测试异常处理程序或报告一个错误情况，"就像"这种情况下解释器引起的相同异常；但要注意没有办法能阻止用户代码引发一个不当的错误。

内置的异常类可以创建子类来定义新的异常；程序员应该从Exception类或它的一个子类而不是从BaseException派生新的异常。有关定义异常详细信息可以访问Python 教程中的用户定义的异常。

以下的异常只用作其它异常的基类。

*exception*BaseException 所有内建的异常的基类。它并不意味用户定义的类应该直接继承它（为此，请使用Exception）。如果在该类的一个实例上调用str()或unicode()，则返回该实例的参数的表示，没有参数时返回空字符串。

版本2.5中新增。

args 异常构造函数的参数元组。有些内建的异常（如IOError）期望一定数量的参数并为此元组的元素分配特殊的含义，而其它异常的调用通常只需要一个单一的字符串来提供一条错误消息。

*exception*Exception 所有内置的、 非系统退出异常是从该类派生的。此外应该从该类派生所有用户定义的异常。

2.5 版本中的更改：更改为从BaseException继承。

*exception*StandardError 除StopIteration、GeneratorExit、KeyboardInterrupt和SystemExit以外的所有内置异常的基类。StandardError本身继承自Exception。

*exception*ArithmeticError 各种算术错误引发的内置异常的基类：OverflowError、ZeroDivisionError、FloatingPointError。

*exception*BufferError 当缓冲区相关的操作无法执行时引发。

*exception*LookupError 当用于映射或序列的键或索引无效时引发的异常的基类：IndexError、KeyError。可以直接通过codecs.lookup()引发。

*exception*EnvironmentError Python系统以外发生的异常的基类：IOError、OSError。当用2元组创建这种类型的异常时，第一项可以通过实例的errno属性访问（它被假设为一个错误编号），第二个项目是可通过strerror属性访问（它通常与错误消息关联）。元组本身也是可用的args属性上的。

1.5.2 版中新增。

EnvironmentError异常以3元组实例化时，前两项的访问和上面一样，第三项可以通过filename属性访问。然而，对于向后兼容性， args属性包含仅 2 元第一次的两个构造函数参数。

当该异常以非3个参数创建时，filename属性为None。当实例不是以2个或3个参数创建时，errno和strerror属性也为None。在最后一种情况下， args以一个元组的形式包含构造函数的原样参数。

以下是实际中会引发的异常。

*exception*AssertionError 当assert语句将失败时引发。

*exception*AttributeError 当属性引用 （请参见属性引用） 或分配失败。（当对象不支持属性引用或根本属性分配时， TypeError将引发。）

*exception*EOFError 当 （input ()或raw_input()） 的内置函数之一点击文件结尾 (EOF) 条件下没有读取任何数据时引发。（注： file.read()和file.readline()的方法返回一个空字符串，当他们击中 EOF。）

*exception*FloatingPointError 提出当浮动点操作将失败。此异常其始终定义，但可以当Python 配置与时的情况下，才会引发— — fpectl 与选项或WANT_SIGFPE_HANDLER符号在pyconfig.h文件中定义。

*exception*GeneratorExit 当调用一种发电机的close ()方法时引发。它直接继承而不是StandardErrorBaseException，因为它是从技术上讲不是一个错误。

新版本 2.5 中的。

2.6 版本中的更改：更改为从BaseException继承。

*exception*IOError O 相关的原因，例如，"未找到文件"或"磁盘已满"（如print语句、 内置open()函数或文件对象的方法） 的 I/O 操作失败时引发。

此类是从EnvironmentError派生的。异常实例属性请参阅上文讨论的详细信息。

2.6 版本中的更改：改变socket.error把此作为基类。

*exception*ImportError 当import语句无法找到模块定义或者时从......引发导入未能找到要导入的名称。

*exception*IndexError 序列下标超出范围时引发。（切片索引会被自动截断落在允许的范围内；如果索引不是一个普通整数，则引发TypeError ）。

*exception*KeyError 在现有的键的集合中找不到 （词典） 的映射键时引发。

*exception*KeyboardInterrupt 当用户点击中断键 （通常控制 C或删除） 时引发。在执行期间，从理论上进行定期检查的中断。中断类型，当一个内置函数的 input ()或raw_input()等待输入还引发此异常。异常继承BaseException， 不意外地被捕获的异常的代码捕捉，从而防止该解释器退出。

2.5 版本中的更改：更改为从BaseException继承。

*exception*MemoryError 当操作耗尽了内存，但情况仍可能获救 （通过删除一些对象） 时引发。关联的值是一个字符串，指示什么样的 （内部） 操作耗尽了内存。请注意由于底层内存管理结构 （C 的malloc ()函数）， 口译员未必能够完全恢复从这种情况 ；它然而引发异常以便可以打印堆栈回溯，离家出走的程序令的情况下。

*exception*NameError 当找不到本地或全局名称时引发。这仅适用于不合格的名称。关联的值是一条错误消息，其中包括找不到的名称。

*exception*NotImplementedError 此异常是从RuntimeError派生的。用户定义基类中抽象方法应引发异常，当他们要求派生的类重写该方法。

在 1.5.2 版本新。

*exception*OSError 此异常是从EnvironmentError派生的。它被提出当一个函数返回与系统相关的错误 （不是非法的参数类型或其他附带的错误）。Errno属性是从errno， 一个数字错误代码， strerror属性是相应的字符串，将印的 C 函数perror ()。请参阅模块errno，其中包含由底层操作系统定义的错误代码的名称。

对于涉及 （如chdir()或作用是：） 的文件系统路径的异常，异常实例将包含三个属性，文件名，这是传递给函数的文件的名称。

在 1.5.2 版本新。

*exception*OverflowError 太大而无法表示算术运算的结果时引发。长整数 （这比放弃，而是会引起MemoryError)，与普通的整数，而是返回一个长整数的大多数操作，就不能发生这种情况。缺乏标准化的浮动点中的异常处理 C，最浮点运算也不会检查。

*exception*ReferenceError 弱引用代理，由weakref.proxy()函数中，创建用于访问属性的指涉后它已被垃圾回收, 时，将引发此异常。弱引用的详细信息，请参阅weakref模块。

新版本 2.2 中的：以前称为weakref。ReferenceError异常。

*exception*RuntimeError 这不会在任何其他类别中检测到错误时引发。关联的值是一个字符串，指示什么精确地走错了。

*exception*StopIteration 提出的迭代器的next ()方法，信号说那里是没有进一步的值。这是例外，而不是StandardError，从推导出来的因为这被认为是一种不在其正常的应用程序中的错误。

新版本 2.2 中的。

*exception*SyntaxError 当解析器遇到语法错误时引发。阅读的最初的脚本或标准输入 （也是以交互方式） 时，这可能发生在exec语句中调用内置函数eval ()或input ()，或导入的语句中。

此类的实例有属性文件名、空格符、偏移量和更容易访问详细信息的文本。str()的异常实例返回唯一的消息。

*exception*IndentationError 与相关的缩进不正确的语法错误的基类。这是SyntaxError的一个子类。

*exception*TabError 当压痕包含制表符和空格的使用不一致时引发。这是IndentationError的一个子类。

*exception*SystemError 当译员发现内部错误，但情况看起来不那么严重，使它不得不放弃所有希望时引发。关联的值是一个字符串，指示发生错误 （在底层的角度来说)。

你应该向作者或维护者你 Python 解释器报告。一定要报告的 Python 解释器版本 （sys.version ； 它也印在 Python 的交互式会话开始时） ，确切的错误消息 （异常的关联值） ，如果可能的程序源代码，触发错误。

*exception*SystemExit 由sys.exit()函数引发此异常。当它不处理时，Python 解释器退出 ；打印没有堆栈回溯。如果关联的值是一个普通整数，它指定系统退出状态 （传递给 C 的exit()函数） ；如果它是None，退出状态为零 ；如果它有另一种类型 （如字符串） ，该对象的值印，退出状态是一个。

实例具有的属性代码设置为 （默认情况下没有） 拟议的退出状态或错误消息。此外，此异常派生直接从BaseException和不StandardError，因为它不是技术上的错误。

对sys.exit()的调用被翻译成异常，以便可以执行清理处理程序 （最后条款try语句） ，并且，以便调试器可以执行一个脚本不运行失控的风险。如果它是绝对有必要退出立即 （例如，在子进程后对os.fork()的调用） ，可以使用os._exit()函数。

异常继承BaseException而不是StandardError或异常，以便不意外地被捕获的异常的代码。这允许将异常正确传播起来并导致该解释器退出。

2.5 版本中的更改：更改为从BaseException继承。

*exception*TypeError 当操作或函数应用于不合适类型的对象时引发。关联的值是字符串，它提供有关类型不匹配的详细信息。

*exception*UnboundLocalError 当提及到一个本地变量在函数或方法，但没有值已绑定到该变量时引发。这是NameError的一个子类。

在 2.0 版中的新。

*exception*UnicodeError 有关 Unicode 编码或解码错误出现时引发。它是ValueError的一个子类。

UnicodeError具有描述编码或解码错误的属性。例如， err.object[err.start:err.end]给出了所编解码器的失败与特定的无效输入。

encoding 引发错误编码的名称。

reason 描述特定的编码解码器错误的字符串。

object 编解码器对象试图进行编码或解码。

start 第一个索引的对象中的数据无效。

end 后最后一个无效的数据对象中的索引。

在 2.0 版中的新。

*exception*UnicodeEncodeError 编码过程中出现的一个 Unicode 相关的错误时引发。它是UnicodeError的一个子类。

新版本 2.3。

*exception*UnicodeDecodeError 解码过程中出现的一个 Unicode 相关的错误时引发。它是UnicodeError的一个子类。

新版本 2.3。

*exception*UnicodeTranslateError 在翻译过程中出现的一个 Unicode 相关的错误时引发。它是UnicodeError的一个子类。

新版本 2.3。

*exception*ValueError 当内置操作或功能接收具有正确的类型，但不正确的值，这样一种说法，这种情况不描述的更精确异常 （如IndexError时引发。

*exception*VMSError 仅在 VM 上可用。当 VM 特定错误时引发。

*exception*WindowsError Windows 特定的错误发生时，或者是错误号码不对应errno值时引发。Winerror和strerror的值创建的GetLastError()和FormatMessage()的功能，从 Windows 平台 API 的返回值。Errno值将winerror值映射到相应的errno.h值。这是OSError的一个子类。

在 2.0 版中的新。

2.5 版本中的更改：以前的版本放入errno的GetLastError()代码。

*exception*ZeroDivisionError 当一个司或取模操作的第二个参数是零时引发。关联的值是一个字符串，指示的操作数和操作的类型。

以下异常作为警告类别使用；详细信息请参阅warnings模块。

*exception*Warning 警告类的基类。

*exception*UserWarning 对于由用户代码生成警告的基类。

*exception*DeprecationWarning 有关已弃用功能的警告的基类。

*exception*PendingDeprecationWarning 警告有关将在未来被否决的功能的基类。

*exception*SyntaxWarning 警告有关可疑语法的基类

*exception*RuntimeWarning 基类有关可疑的运行时行为的警告。

*exception*FutureWarning 基类构造，它们会在未来发生语义变化有关的警告。

*exception*ImportWarning 导入模块中可能错误的警告的基类。

2.5 版中新增。

*exception*UnicodeWarning Unicode相关的警告的基类。

2.5 版中新增。

6.1. 异常的层次结构

内置异常的类层次结构是：

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

7. String Services

The modules described in this chapter provide a wide range of string manipulation operations.

In addition, Python's built-in string classes support the sequence type methods described in the *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange* section, and also the string-specific methods described in the *String Methods* section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. Also, see the re module for string functions based on regular expressions.

8. Data Types

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, dict, list, set (which along with frozenset, replaces the deprecated sets module), and tuple. The str class can be used to handle binary data and 8-bit text, and the unicode class to handle Unicode text.

9. Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the decimal module supports exact representations of decimal numbers.

10. File and Directory Access

The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

11. Data Persistence

The modules described in this chapter support storing Python data in a persistent form on disk. The pickle and marshal modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings. The bsddb module also provides such disk-based string-to-string mappings based on hashing, and also supports B-Tree and record-based formats.

13. File Formats

The modules described in this chapter parse various miscellaneous file formats that aren鈥檛 markup languages or are related to e-mail.

14. Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation.

Hardcore cypherpunks will probably find the cryptographic modules written by A.M. Kuchling of further interest; the package contains modules for various encryption algorithms, most notably AES. These modules are not distributed with Python but available separately. See the URL http://www.pycrypto.org for more information.

15. Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well.

16. Optional Operating System Services

17. Interprocess Communication and Networking

The modules described in this chapter provide mechanisms for different processes to communicate.

Some modules only work for two processes that are on the same machine, e.g. signal and subprocess. Other modules support networking protocols that two or more processes can used to communicate across machines.

18. Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the Internet.

20. Internet Protocols and Support

The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module socket, which is currently supported on most popular platforms.

26. Debugging and Profiling

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs.

28. Python Runtime Services

Python 2 语言参考

这份参考手册讲述该语言的语法和“核心语义”。它虽然简洁，但是力求准确和全面。不是语言必要部分的内建对象类型、内建函数和模块的语义在Python标准库中讲述。关于语言的非正式的介绍，请参阅Python教程。对于C和C++程序员，还存在另外两个手册：扩展和嵌入Python解释器讲述关于如何编写Python扩展模块的高级话题，Python/C API参考手册详细地讲解对C/C++程序员可用的接口。

1. 简介

这份参考手册讲述Python编程语言。它并不打算作为一个教程。

除了语法和词法分析，在尽可能精确的同时，对所有内容我选择使用英语而不是形式化的说明。这应该使得这份文档对一般读者更容易理解, 但将导致某些地方意义不明确。因此, 如果你来自火星并试图仅仅通过这份文档重新实现 Python, 你可能不得不猜测一些东西, 其实事实上你将可能最终实现一种完全不同的语言。另一方面, 如果你正在使用 Python 并想知道该语言某一方面明确的规则, 你一定能在这里找到它们。如果你想要看该语言更正式的定义, 或许你可以自愿献出你的时间来写一份 — 那还不如用来发明一台克隆机器 :-)。

加入太多的实现细节到语言参考文档里是危险的 — 实现可能改变，同一语言的其它实现可能以不同的方法工作。另一方面，CPython是当前唯一一个广泛使用的Python实现（虽然存在其它实现），它的某些怪异的地方有时候是值得提及的，尤其是强加了额外限制的实现。因此，你将会发现散布在整个文档中的简短的“实现说明”。

每一种Python实现都伴随着若干内建和标准模块。这些文档位于*Python标准库*。一些内建的模块以重要的方式与语言的定义交互时会有所提及。

1.1. 各种实现

尽管已有一个目前最为流行的 Python 实现, 但还是有一些其它的实现, 它们对不同的用户有着特别的吸引力。

已知的实现包括：

CPython这是Python初始的以及维护得最好的实现，使用C编写。新的语言特性一般会最先在这里出现。Jython用Java 实现的Python。这个实现可以作为脚本语言在Java应用中使用，或者可以用来利用Java类库来创建应用。它也经常被用来为Java库创建测试。更多的信息可以在Jython网站上找到。用于.NET的Python这个实现实际上使用了CPython实现, 但是是一个.NET托管的应用程序，并使得.NET类库可以使用。它由Brian Lloyd创建。更多信息请参阅.NET版Python的主页。IronPython.NET 版的另外一种Python。与Python.NET不同，这是一个完整的Python实现，它产生IL并且直接把Python代码编译成.NET程序集。它由Jython 的初始创建者Jim Hugunin 创建。更多信息请参阅IronPython 网站。PyPy完全用Python 写的一种Python 实现。它支持一些在其它实现中没有的高级特性，例如无栈支持和JIT（即时）编译器。该项目的目标之一是，鼓励通过使得改变解释器更简单来试验语言本身（因为它是用Python 写的）。更多信息可以访问 PyPy 项目的主页。 这里的每一个实现都会与这份手册里讲述的语言在某些方面有所不同, 或者引入超出标准Python 文档内容的特殊信息。请参阅特定实现的文档以确定关于你正在使用的特定实现，你还需要了解些什么。

1.2. 语法符号

词法分析和语法的描述使用一种修改过的BNF语法符号。它使用以下风格的定义：

```
name      ::=  lc_letter (lc_letter | "_")*
lc_letter ::=  "a"..."z"
```

第一行是说name是一个lc_letter，后面跟着一个零个或多个lc_letter和下划线组成的序列。接着，一个lc_letter是'a'到'z'之间任意一个单个字符。（这个规则事实上就是该文档中词法和语法规则中的名称的定义方式。）

每条规则以一个名字（这条规则定义的名字）和::=开始。竖线(|)用于分隔可选的项；它是该语法符号中绑定性最弱的操作符。星号*()*表示前面项目的零个或多个重复；类似地, 加号*(+)*表示一个或多个重复, 而方括号*([])*表示里面的内容出现零次或一次（换句话说, 方括号中的内容是可选的）。和+操作符的绑定性最强；圆括号用于分组。字符串字面值由引号引起来。空格只对分隔标识符有意义。规则通常包含在单独的一行中；具有许多可选项的规则可能会在第一行之后，每一行以一个竖线开始。

在词法定义中（如上面的例子），还使用了两个额外的约定：三个点号分隔的两个字符表示给出的范围内（包括这两个字符）的任何一个单个ASCII字符。尖括号(<...>)中的内容表示不是定义的符号的正式描述；例如，如果需要这可以用来描述‘控制字符’的概念。

虽然使用的语法符号几乎完全一样，词法和语法定义之间的含义有一个巨大的差异：词法定义工作在输入的单个字符上，而语法定义工作在由词法分析生成的标识符流上。下一章（“词法分析”）中使用的所有BNF都是词法定义；再往后的几章是语法定义。

2. 词法分析

Python程序由解析器读取。输入到解析器中的是由词法分析器生成的词符流。本章讲述词法分析器如何把一个文件拆分成词符。

Python程序的文本使用7比特ASCII字符集。

2.3版中新增：可以使用编码声明指出字符串字面值和注释使用一种不同于ASCII的编码。

为了和旧的版本兼容，如果发现8比特字符，Python只会给出警告。修正这些警告的方法是声明显式的编码，或者对非字符的二进制数据字节使用转义序列。

运行时的字符集取决于与程序连接的I/O设备，但通常都是ASCII的超集。

未来兼容性的注意事项：可能会假设8比特字符的字符集是ISO Latin-1（ASCII字符集的超集，覆盖了大部分使用拉丁字母的西方语言），但是在未来Unicode文本编辑器可能变得通用。这些编辑器通常使用UTF-8编码，它也是ASCII的超集，但是序数在128-255之间的字符的用法非常不一样。关于这个主题还没有一致的意见，假设是Latin-1或UTF-8都不明智，即使当前的实现似乎倾向于Latin-1。源文件的字符集和运行时的字符集都适用。

2.1. 行结构

一个Python程序被分割成若干逻辑行。

2.1.1. 逻辑行

逻辑行的结束由NEWLINE词符表示。语句不能跨越逻辑行的边界除非语法允许NEWLINE（例如，复合语句的语句之间）。通过遵循显式或隐式的行连接规则，一个逻辑行由一个或多个物理行构成。

2.1.2. 物理行

一个物理行是一个被行结束序列终止的字符序列。在源文件中，任何标准平台的行终止序列都可以使用 - Unix方式使用ASCII的LF（换行），Windows方式使用ASCII序列CR LF(回车和换行)，旧的Macintosh方式使用ASCII的CR（回车）字符。

在嵌入Python时，对于换行符源代码字符串应该使用标准C的习惯传递给Python API（代表ASCII LF的\n字符是行终止符）。

2.1.3. 注释

注释以非字符串字面值中的井号字符(#)开始，在物理行的末尾结束。除非引起隐式的行连接规则，否则注释意味着逻辑行的结束。语法会忽略注释；它们不是词符。

2.1.4. 编码声明

如果Python脚本的第一行或者第二行的注释匹配正则表达式coding[=:]\s*([-\w.]+)，那么这行注释将作为编码声明处理；该表达式的第一个分组指出源文件编码的名字。建议的表达式形式是

```
# -*- coding: <encoding-name> -*-
```

它也能被GNU Emacs识别，或者

```
# vim:fileencoding=<encoding-name>
```

它能被Bram Moolenaar的VIM识别。除此之外，如果文件开始几个字节是UTF-8的字节顺序标记('\xef\xbb\xbf')，声明的文件编码将是UTF-8（这个特性也被微软的**notepad**和其它编辑器支持。）

如果声明了编码，那么编码的名字必须能被Python识别。编码将用于所有的词法分析，特别是寻找字符串的结束，和解释Unicode字面值的内容。字符串字面值会被转换成Unicode来做语法分析，然后在解释开始之前被转换回它们初始的编码。编码声明必须出现在它自己单独的一行上。

2.1.5. 显式的行连接

两个或多个物理行可以使用反斜杠字符()连接成一个逻辑行，方式如下：当一个物理行以一个不是字符串字面值或注释中的反斜杠结束时，它会和接下来的一行连接形成一个单独的逻辑行，反斜杠和后面的换行符会被删掉。例如：

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

以反斜杠结束的行不能带有注释。反斜杠不能延续注释。反斜杠不能延续除了字符串字面值以外的词符（即不是字符串字面值的词符不能使用反斜杠分割跨多个物理行）。一行中位于字符串字面值以外其它地方的反斜杠是非法的。

2.1.6. 隐式的行连接

圆括号、方括号以及花括号中的表达式可以分割成多个物理行而不使用反斜杠。例如：

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

隐式的续行可以带注释。续行的缩进不重要。允许空白的续行。隐式的续行之间没有NEWLINE词符。隐式的续行也可以发生在三引号的字符串中（见下面）：在这种情况下它们不可以带注释。

2.1.7. 空白行

只包含空格符、制表符、换页符和注释的逻辑行会被忽略（即不会有NEWLINE词符生成）。在交互式输入语句时，空白行的处理可能不同，这取决于read-eval-print循环的实现。在标准实现中，一个完全的空白逻辑行（即一个不只是包含空格或注释的逻辑行）会终止多行语句。

2.1.8. 缩进

逻辑行开始的前导空白（空格和制表符）用来计算行的缩进层级，然后用它决定语句的分组。

首先，制表符被替换（从左到右）成一至八个空格，这样包括替换后的字符的总数是八的整数（这是为了和Unix 使用一样的规则）。非空白字符之前的空格总数决定该行的缩进。缩进不可以使用反斜杠分割成多个物理行；直到第一个反斜杠处的空白决定了缩进。

跨平台兼容性的注意事项：由于非UNIX平台上的文本编辑器的天性，在一个源文件中缩进混合使用空格和制表符是不明智的。还应该注意到不同的平台可能明确地限定最大缩进的层级。

行的开始可能会出现换页符；它将被上述缩进的计算忽略。在前导空白其它地方出现的换页符的作用没有定义（例如，它们可能会重置空格的数量为零）。

连续行的缩进层级用于生成INDENT和DEDENT词符，这个过程使用了栈，如下所述。

在读入文件第一行之前, 一个零被压入堆栈中；它将再也不会被弹出。压入堆栈中的数字将永远从底部往顶部增长。在每一个逻辑行的开始，该行的缩进层级会与栈顶比较。如果相等，什么都不会发生。如果大于栈顶，将其压入栈，并生成一个INDENT词符。如果小于栈顶，它必须是堆栈中已存在的数字中的一个；栈中所有大于它的数都将被弹出，并且每个弹出的数字都生成一个DEDENT词符。到达文件尾时，栈中剩下的每一个大于零的数字也生成一个DEDENT词符。

这儿是一个正确缩进的Python代码片段的例子（虽然有点乱）：

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

下面的例子展示了各种缩进错误：

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

（事实上；前三个错误是由解析器发现的；仅仅最后一个错误是由词法分析器找到的 -- returnr 的缩进层级与堆栈中弹出的数字没有匹配的层级。

2.1.9. 词符之间的空白

除非位于逻辑行起始处或者字符串字面值当中，空格、制表符和换页符这些空白字符可以等同地用于分隔词符。空白仅当两个词符连接在一起可以理解成一个不同的词符时才需要（例如，ab是一个词符，但a b是两个词符）。

2.2. 其它的词符

除了NEWLINE、INDENT和DEDENT，还存在以下几类词符：标识符、关键字、字面值、操作符和分隔符。空白字符（前面讨论的断行符除外）不是词符，而是用于分隔词符。有歧义存在时，词符由形成合法词符的最长字符串组成（自左向右读取）。

2.3. 标识符和关键字

标识符（也称为名字）由以下词法定义描述：

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

标识符长度没有限制。区分大小写。

2.3.1. 关键字

以下标识符用作保留字，或者叫做语言的关键字，并且不能作为普通的标识符使用。它们必须像下面那样准确拼写：

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

版本2.4 中的变化：None成为一个常量并且被编译器识别为内建对象None的名字。尽管不是关键字，你也不可以给它赋值一个不同的对象。

版本2.5 中的变化：使用as和with作为标识符会引发警告。要使用它们作为关键字，需启用with_statement这个未来特性。

版本2.6 中的变化：as和with成为真正的关键字。

2.3.2 保留类别的标识符

有几种特定类别的标识符（关键字除外）具有特殊的含义。这些类别有标志性的模式就是开始和尾部的下划线：

不会被*frommoduleimport*导入。这个特殊的标识符用于在交互式解释器中存储上一次计算的结果；它存储在**builtin**模块。不在交互式模式时，_没有特别的含义且是未定义的。参看*import* 语句一节。

注意

名字_经常用于国际化；关于这个惯例的更多信息请参考gettext模块的文档。

*系统定义的名字。这些名字由解释器及其实现（包括标准库）定义。这些系统定义的名字在特殊方法的名字一节和其它地方讨论。未来版本的Python 可能会定义更多的系统名字。无论什么情况，任何不遵守明确的文档使用说明的*使用，都可能会带来破坏而没有警告。__类私

有变量。这种类别的名字，在类定义的语境中使用时，会被使用一种变形的形式重写以避免基类和继承类的“私有”属性之间的冲突。参考[标识符（名称）*](#)一节。

2.4. 字面值

字面值是某些内建类型常量的标写法。

2.4.1. 字符串字面值

字符串字面值由以下词法定义描述：

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
                     | "b" | "B" | "br" | "Br" | "bR" | "BR"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''"
                     | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | escapeseq
longstringitem  ::=  longstringchar | escapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
escapeseq       ::=  "\" <any ASCII character>
```

上面产生式中没有表示出来一个的语法限制是， 在stringprefix与其余字符串字面值之间不允许出现空白字符。字符集由编码声明定义；如果源文件中没有指定编码声明，则为ASCII；参考编码声明一节。

用简单的中文来描述就是：字符串字面值可以包含在配对的单引号(')或双引号(")中。它们也可以包含在配对的三个单引号或双引号组中（这些字符串一般称为三引号字符串）。反斜杠()用于转义具有特殊意义的字符，例如换行、反斜杠本身或者引号。字符串字面值可以加一个前缀字母'r' 或者'R'；这些字符串称为原始字符串 并且使用不同的规则解释反斜杠转义的序列。前缀'u'或'U'使得字符串成为一个Unicode字符串。Unicode 字符串使用由Unicode 协会和ISO 10646定义的Unicode 字符集。Unicode 字符串中有效的一些额外转义序列会在下面描述。前缀'b'或'B'在Python 2中被忽略；在Python 3中， 它表示那个字面值应该是一个字节型字面值（例如，用2to3自动转换代码的时候）。前缀'u'或'b'后面可以跟随一个前缀'r'。

在三引号字符串中，没有转义的换行和引号是允许的（并且会被保留）， 除非三个未转义的引号终止了字符串。（引号指用于开始字符串的字符，例如' 或"。）

除非出现前缀'r' 或'R'，否则字符串中的转义序列依照类似标准C 使用的规则解释。可识别的转义序列有：

转义序列	含义	说明
\newline	忽略	
\	反斜杠 ()	
\'	单引号 (')	
\"	双引号 (")	
\a	ASCII响铃(BEL)	
\b	ASCII退格(BS)	
\f	ASCII换页(FF)	
\n	ASCII换行(LF)	
\N{name}	Unicode数据库中名为*name*的字符(Unicode only)	
\r	ASCII回车(CR)	
\t	ASCII水平制表(TAB)	
\uxxxx	16位的十六进制值为*xxxx*的字符(Unicode only)	(1)
\Uxxxxxxxx	32位的十六进制值为*xxxx*的字符(Unicode only)	(2)
\v	ASCII垂直制表(VT)	
\ooo	八进制值为*ooo*的字符	(3,5)
\xhh	十六进制值为*hh*的字符	(4,5)

说明：

1. 形成代理对一部分的代码单元可以使用这种转义序列编码。
2. 任何Unicode字符都可以用这种方式编码，但如果Python 编译成使用16比特代码单元（默认行为），位于基本多语言平面(BMP)之外的字符将用代理对编码。
3. 与标准C 一样，最多接受三个八进制数字。
4. 与标准C 不同，要求两个精确的十六进制数字。
5. 在字符串字面值中, 十六进制和八进制转义字符表示具有给定值的字节；没有必要再用用那个字节编码源字符集中的字符。在Unicode字面值中，这些转义字符表示一个具有给定值的Unicode字符。

与标准C 不同, 所有不能识别的转义序列都会保留留在字符串中维持不变，例如，反斜杠会保留在字符串串中。（这个行为在调试的时候特别有用：如果敲错一个转义序列，输出的结果可以很容易看出是有问题的。）同样要注意，上面表格中标记为“(Unicode only)”的转义序列，在非Unicode字符串字面值中属于不能识别的类别。

当出现前缀'r' 或'R'时，紧跟在反斜杠后面的字符会包含在字符串中不变，并且所有的反斜杠都会保留在字符串中。例如，字符串字面值r"\n" 由两个字符组成：一个反斜杠和一个小写的'n'。字符串的引号可以用反斜杠转义，但是反斜杠会保留在字符串中；例如，r"\"" 是一个由

两个字符组成的合法的字符串字面值：一个反斜杠和一个双引号；r"\" 不是一个合法的字符串字面值（即使原始字符串不能以奇数个反斜杠结束）。特别地, 原始字符串不能以一个反斜线结束（因为反斜杠会转义随后的引用字符）。还要注意后面紧跟着换行符的反斜杠被解释为字符串中的两个字符，而不是作为续行处理。

如果前缀'r' 或者'R' 和前缀'u' or 'U'一起使用，那么转义序列uXXXX 和UXXXXXXXX 会被处理而其它所有反斜杠会保留在字符串中。例如，字符串字面值ur"\u0062\n" 有三个Unicode 字符组成：'拉丁小写字母b'、'反斜杠'和'拉丁小写字母n'。反斜杠可以用前面的反斜杠转义；然而，两个都会保留在字符串中。结果，转义序列\uXXXX只有在奇数个反斜杠的时候才能识别。

2.4.2. 字符串字面值的连接

多个相邻的字符串字面值（由空白分隔），可能使用不同的引用习惯，是允许的且含义是把它们连接起来。因此，"hello"'world'等同于"helloworld" 。这个特性能够用于减少需要的反斜杠的数目以方便地把很长的字符串分成几行，或者甚至给字符串的某些部分加上注释，例如：

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

注意，这个特性在语法层面上定义，但是在编译的时候实现。'+'运算符用于连接字符串表达式必须在运行的时候。还需要注意，字面值的连接可以为每个部分使用不同的引号风格（即使是混合原始字符串和三引号字符串）。

2.4.3. 数值字面值

有四种类型的数值字面值：普通整数、长整数、浮点数和虚数。没有复数字面值（复数字面值可以由一个实数和一个虚数相加形成）。

注意数值字面值不包括符号；像-1这样的短语实际上是由一元运算符'-'和字面值1组成的表达式。

2.4.4. 整数和长整数字面值

整数和长整数字面值可以用以下词法定义描述：

```
longinteger    ::=  integer ("l" | "L")
integer        ::=  decimalinteger | octinteger | hexinteger | bininteger
decimalinteger ::=  nonzerodigit digit* | "0"
octinteger     ::=  "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger     ::=  "0" ("x" | "X") hexdigit+
bininteger     ::=  "0" ("b" | "B") bindigit+
nonzerodigit   ::=  "1"..."9"
octdigit       ::=  "0"..."7"
bindigit       ::=  "0" | "1"
hexdigit       ::=  digit | "a"..."f" | "A"..."F"
```

大于最大可表示普通整数的普通整数字面值（例如，2147483647）会被当作长整数。[1]长整数字面值大小没有限制，除了可用内存的容量。

一些普通整数字面值（第一行）和长整数字面值（第二行和第三行）的例子：

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

2.4.5. 浮点数字面值

浮点数字面值由以下词法定义描述：

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

注意浮点数的整数部分和指数部分可能看上去像八进制数，但仍然用十进制解释。例如，077e010是合法的，它和77e10表示同一个数。浮点数字面值允许的范围依赖于具体的实现。一些浮点数字面值的例子：

```
3.14    10.    .001    1e100    3.14e-10    0e0
```

注意数值字面值不包括符号；像-1这样的短语实际上是由一元运算符'-'和字面值1组成的表达式。

2.4.6. 虚数字面值

虚数字面值由以下词法定义描述：

```
imagnumber ::=  (floatnumber | intpart) ("j" | "J")
```

虚数字面值生成一个实部为0.0 的复数。复数由一对具有相同取值范围的浮点数表示。若要创建一个实部非零的复数，可以给它相加一个浮点数，例如，(3+4j)。一些虚数字面值的例子：

```
3.14j   10.j    10j     .001j   1e100j  3.14e-10j
```

2.5. 操作符

以下词符是操作符：

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

比较操作符<>和!=是相同操作符的两个可选的拼写。推荐使用!=拼写，<>已经过时。

2.6. 分隔符

以下词符用作文法中的分隔符：

```
(       )       [       ]       {       }      @
,       :       .       `       =       ;
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

句号也可以出现在浮点数和虚数字面值中。一个连续三个句号的序列在切片中具有省略号这样特殊的含义。列表的第二部分，即参数化赋值运算符，在词法上是作为分隔符处理，但也执行运算。

以下可打印ASCII字符作为其它词符的一部分有着特殊的含义，或者对于词法分析器具有重要作用：

```
'       "       #       \
```

以下可打印ASCII字符在Python中没有使用。它们出现在字符串字面值和注释之外是无条件的一个错误：

```
$       ?
```

脚注

| [1] | 在Python 2.4之前的版本中，在最大的可表示普通整数和最大的无符号32位数字4294967296（在一台使用32位算术的机器上）范围之间八进制和十六进制字面值会通过减去4294967296作为负的普通整数。 | |-----|-----|

3. 数据模型

3.1 对象、值和类型

对象是Python对数据的抽象。Python程序中所有数据都由对象或对象之间的关系表示。（合理且与冯.诺依曼的“存储程序计算机”模型一致，代码也由对象表示。）

每个对象都有一个ID，一个类型和一个值。对象一旦建立，它的*ID*永远不会改变；你可以认为它是该对象在内存中的地址。‘is’操作符比较两个对象的ID；id()函数返回一个表示对象ID 的整数（当前实现为对象的地址）。对象的类型也是不可变的。[1]对象的类型决定了对象支持的操作（例如,“它有长度吗?”）同时也定义了该种类型对象的可能的值。type()函数返回对象的类型（它本身也是一个对象）。某些对象的值可以改变。值可以改变的对象称为可变的；一旦建立，值就不可以改变的对象称为不可变的。（包含可变对象引用的不可变容器对象在可变对象改变时是可以改变的；但容器仍被看作是可变的, 因为它所包含的对象集合是不能变的。所以不可变对象与值不可变不是完全一样的，它更加微妙。）一个对象的可变性由它的类型决定；例如，数值、字符串和元组是不可变的，而字典和列表是可变的。

对象不可以显式地销毁；但是当它们不可用时可能被当作垃圾回收。具体的实现可以推迟垃圾回收或完全忽略它 — 这是垃圾回收如何实现的质量问题，只要依然能访问的对象不被回收。

CPython 实现细节：CPython 当前使用引用计数机制与（可选的）循环连接垃圾延迟检测机制，一旦对象变得不可访问，它将收集其中大部分，但是不保证收集包含循环引用的垃圾。参考gc 模块的文档可以获得控制循环垃圾回收的信息。其它实现的行为与之不同并且CPython 的实现将来也可能会变化。不要依赖对象不可访问后会立即终结（例如：永远关闭文件）。

注意使用具体实现的跟踪和调试工具可能会保持正常情况下可以回收的对象一直存活。还要注意使用‘try...except’语句捕获异常也可能保持对象一直存活。

有些对象包含“外部”资源的引用，例如打开的文件或窗口。可以理解在对象被当作垃圾回收时这些资源也被释放，但因为垃圾回收不保证一定发生，这样的对象也提供显式的方法释放外部资源，通常是close()方法。强烈建议程序显式关闭这些对象。‘try...finally’语句提供了一个便利的方式来做这件事。

有些对象包含其它对象的引用；它们叫做容器。容器的例子有元组，列表和字典。引用是容器的值的一部分。大多数情况下，当我们谈到一个容器的值时，我们是指值，而不是所包含的对象的ID；然而，当我们谈论容器对象的可变性的时候，就只是指被直接包含的对象的ID。因此，如果一个不可变对象（如元组）包含了一个可变对象的引用，那么当这个可变对象的值改变时它的值也发生改变。

对象的类型几乎影响对象的所有行为。在某种意义上甚至重要到影响对象的识别：对于不可变对象，计算新值的运算符可能实际上返回的是一个已存在的具有相同类型和值的对象的引用，而对于可变对象，这是不允许的。例如，在a=1;b=1之后，a 和b可能是或者可能不是引用同一个值为1的对象，这依赖于实现，但c=[];d=[]之后，c 和d可以保证是引用两个不同的、唯一的、新创建的空列表。（注意c=d=[]是把相同的对象赋给c 和d。）

3.2 标准类型的层次结构

以下是一份Python 内建类型的清单。扩展模块（无论是用C、Java还是用其它语言编写，依赖于具体实现）可以定义额外的类型。未来版本的Python 可能在这个类型层次机构中增加其它类型（例如：有理数、高效存储的整数数组，等等）。

下面有些类型的描述包含一个列出“特殊属性”的段落。这些属性提供访问具体的实现而不是作为一般的目的使用。它们的定义在未来可能会改变。

None 这种类型只有一个值。。只有一个对象具有这个值。这个对象通过内建名字None访问。它在许多情况下用来表示没有值，例如，没有显式返回任何内容的函数会返回它。它的真值为假。

NotImplemented 这种类型只有一个值。只有一个对象具有这个值。这个对象通过内建名字NotImplemented访问。如果数值方法和复杂的比较方法没有为提供的操作数实现某种运算，它们可能返回这个值。（解释器会尝试反射的操作，或者其它退化的操作，依赖于具体的运算符。）它的真值为真。

Ellipsis 这种类型只有一个值。只有一个对象具有这个值。这个对象通过内建名字Ellipsis访问。它用于指示切片中出现的...语法。它的真值为真。

numbers.Number 它们由数值字面量生成或者由算术运算符和内建的算术函数作为结果返回。数值对象是不可变的；一旦创建，它们的值永远不会改变。Python 的数值和数学上的数字关系当然是非常密切的，但受到计算机数值表达能力的限制。

Python 区分整数、浮点数和复数：

numbers.Integral 它们表示数学上的整数集中的元素（包括正数和负数）。

有三种类型的整数：

普通整数 它们表示在-2147483648 至2147483647 范围之间的数。（这个范围可能会在本地机器字较大的机器更大些，但不会更小。）如果某个运算的结果超出这个范围，结果会以长整数正常返回（在某些情况下，会抛出异常OverflowError）。对于以移位和掩码为目的的运算，整数采用32位或更多位的二进制补码形式，并且不会对用户隐藏任何位。（就是说，所有4294967296个不同的比特组合对应于不同的值）。

长整数 长整数的表示的数值范围没有限制，只受限于可用的（虚拟内存）内存。对于以移位和掩码为目的的运算，长整数采用二进制的形式，负数用二进制补码形式表示，给人的错觉是一个符号位向左无限扩展的字符串。

布尔值 布尔值表示假和真的真值。表示False 和True 的两个对象是仅有的布尔对象。布尔类型是普通整数的子类型，布尔值的行为在几乎所有环境下分别类似0和1，例外的情况是转换成字符串的时候分别返回字符串"False" 或"True"。

整数表示法的规则意在让负数的移位和掩码运算具有最有意义的解释，并且在普通整数和长整数之间转换时具有最少的意外。任何运算，只要它产生的结果在整数域之中，那么在长整数域或混合运算时将产生相同结果。域之间的转换对程序员是透明的。

numbers.Real (float) 这种类型表示机器级别的双精度浮点数。你受底层的机器体系结构（和C或者Java的实现）控制接受的范围和溢出处理。Python不支持单精度浮点数；使用它的原因通常是节省处理器和内存的使用，但是相比Python中对象使用的开销是微不足道的，因此没有必要支持两种浮点数使语言变的复杂。

numbers.Complex 这种类型以一对机器级别的双精度浮点数表示复数。单精度浮点数同样可以用于复数类型。复数z的实部和虚部可以通过只读属性z.real和z.imag获得。

序列 这种类型表示有限的顺序集合，用非负数索引。内建的函数len() 返回序列的元素个数。当序列的长度为*n*时，索引集合包含数字0, 1, ..., *n*-1。序列*a*的元素*i* 的选择使用a[i] 。

序列也支持切片：a[i:j]选择索引*k*满足*i*<=*k*<*j*的所有元素。作为表达式使用的时候，切片是一个相同类型的序列。这隐含着索引会从零开始重新计数。

某些序列还支持带有第三个“步长”参数的“扩展切片”：a[i:j:k] 选择*a* 中所有索引为*x*的 的元素，x=i+n*k*, n>=*0* 且i<=x<j*。

序列依据它们的可变性分为：

不可变序列不可变序列类型的对象一旦创建便不可改变。（如果这个对象包含其它对象的引用，这些引用的对象可以是可变的并且可以改变；然而不可变对象直接引用的对象集合不可改变。）

以下类型是不可变序列：

字符串 字符串的元素是字符。没有单独的字符类型；字符用一个元素的字符串表示。字符表示（至少）8 比特的字节。内建函数chr() 和ord()在字符和表示字节数值的非负整数之间转换。值在0-127 之间的字节通常表示相应的ASCII 值，但是对值的解释由程序决定。字符串数据类型也用于表示字节的数组，例如，保存从文件中读取的数据。

（在原生字符集不是ASCII 的系统上，字符串在内部可以使用EBCDIC 表示，只要函数chr() 和ord() 实现ASCII 和EBCDIC 之间的映射并且字符串的比较保留ASCII 顺序。或者可能有人能够提出一个更好的规则？）

Unicode Unicode 对象的元素是Unicode 编码单元。一个Unicode 编码单元由一个元素的Unicode 对象表示并且可以保持16位或者32位的值表示一个Unicode 序数。（序数的最大值在sys.maxunicode中给出，并依赖Python 在编译的时候是如何配置的）Unicode 对象中可以表示代理对，并被当作两个单独的元素。内建的函数unichr() 和ord()在编码单元和表示定义在Unicode 标准3.0中Unicode 序数的非负整数之间转换。和其它编码之间相互转换可以通过Unicode 方法encode() 和内建的函数unicode()。

元组 元组的元素可以是Python 的任何对象。两个或多个元素的元组由逗号分隔的一连串表达式形成。一个元素的元组（单元素集）可以在一个表达式的后面附加一个逗号形成（一个表达式自身不会形成一个元组，因为圆括号必须可以用来分组表达式）。一个空的元组可以由一个空的圆括号对形成。

可变序列 可变序列在生成之后可以修改。下标和切片表示法可以用于赋值和del (delete)语句的对象。

当前有两种内建的可变序列类型：

列表 列表的对象可以是Python 任何对象。列表由在方括号中放置一个逗号分隔的一连串表达式形成。（注意生成长度为0或1的列表没有特殊的情形。）

字节数组 一个字节数组对象是一个可变的数组。它们由内建的bytearray() 构造函数创建。除了可变性（因此不可哈希），另一方面字节数组提供和不可变字节对象同样的接口和功能。

扩展模块array提供另外一个可变序列类型的例子。

集合类型 这种类型表示无序的、有限的集合，集合中的元素是唯一的、不可变的对象。正因如此，它们不可以被任何下标索引。然而，它们可以迭代，内建函数len()返回集合中元素的个数。集合常见的用途有快速成员关系检测、从序列中删除重复元素和计算数学运算例如交集、并集、差集和对称差集。

集合的元素与字典的键一样，都适用不可变规则。注意，数值类型遵循正常的数值比较规则：如果两个数字相等（例如，1 和1.0），其中只有一个可以包含在集合中。

当前有两种内建的集合类型：

集合 这种类型表示可变的集合。它们由内建函数set() 构造函数创建并可以在此之后通过几种方法修改，例如add()。

固定集合 这种类型表示不可变集合。它们由内建函数frozenset()构造器创建。因为固定集合不可变且可以哈希，它可以作为另外一个集合的元素或者字典的键。

映射 这种类型表示由任意索引集合作索引的有限对象集合。下标表示法a[k] 从映射a 中选择由k 索引的元素；它可以用在表达式中并作为赋值或del语句的目标。内建函数len()返回映射中元素的个数。

当前只有一个内建映射类型：

字典 这种类型表示几乎可以由任何值索引的有限对象集合。不可以作为键的唯一类型是包含列表或者字典或者其它可变类型的值，这些类型通过值而不是对象ID比较，原因是字典的高效实现要求键的哈希值保持常量。注意，数值类型遵循正常的数值比较规则：如果两个数字相等（例如，1 和1.0），那么它们可以互换地使用来索引同一个字典入口。

字典是可变的；它们可以通过{...} 表示法创建（参考*Dictionary的显示*一节）。

扩展模块dbm，gdbm 和bsddb提供另外的映射类型的例子。

可调用类型 这是一种可以使用函数调用操作（参考*Calls*一节）的类型：

用户定义的函数 用户定义的函数对象由函数定义创建（参见函数定义一节）。它调用时参数列表的元素个数应该和函数的形式参数列表相同。

特殊属性：

属性	含义	
docfunc_doc	函数的文档字符串，如果没有就为None。	可写
namefunc_name	函数的名字。	可写
module	函数定义所在的模块名，如果没有就为None。	可写
defaultsfunc_defaults	为具有默认值的参数保存默认参数值的元组，如果没有参数具有默认值则为None。	可写
codefunc_code	表示编译后的函数体的代码对象。	可写
globalsfunc_globals	保存函数全局变量的字典的引用 — 函数定义所在模块的全局命名空间。	只读
dictfunc_dict	支持任意函数属性的命名空间。	可写
closurefunc_closure	None 或者包含函数自由变量绑定的元组。	只读

大部分标有“可写”的属性会检查所赋值的类型。

版本2.4中的变化：func_name 成为如今可写的属性。

版本2.6中的变化：引入双下划线属性**closure**, **code**, **defaults**, 和**globals**作为对应的func_*属性的别名以向前兼容Python 3。

函数对象同样支持获取和设置任意属性，这可以用来附加元数据到函数中。常规属性可以用点号表示法获取和设置。注意当前的实现只在用户定义的函数上支持函数属性。未来可能支持内建函数上的函数属性。

函数定义的额外信息可以从它的代码对象中获取；参见下面内部类型的描述。

用户定义的方法 用户定义的方法将类、类的实例（或者None）和任何可调用对象（通常是一个用户定义的函数）结合起来。

特殊的只读属性：im*self*指类实例对象，*imfunc*指函数对象；*imclass*对于绑定的方法指*imself*的类，对于未绑定的方法指方法所在的类；***doc***指方法的文档（与*imfunc.**doc***相同）；*_name*指方法的名字（与*im_func.__name*一样）；*__module*指方法定义所在模块的名字，如果没有则为None。

版本2.2中的变化：im_self过去指的是定义方法的类。

版本2.6中的变化：为了Python 3向前的兼容性，im*func*也可以使用***func***访问，*imself*可以使用*__self*访问。

方法也支持访问（但不能设置）底层函数对象的任何函数属性。

用户定义的方法对象可能在获取类的一个属性的时候创建（可能通过类的一个实例），如果那个属性是用户定义的函数对象或者一个未绑定的用户方法对象或者一个类方法对象。如果那个属性是用户定义的方法对象，只有类和存储在原始方法对象中的类或其子类相同，才会创建一个新的方法对象；否则，使用初始的方法对象。

如果用户定义的方法是通过从类中获取用户定义的方法创建，它的im_self为None并且方法对象称为

4. 执行模型

4.1. 命名和绑定

名称是对象的引用。名称通过名称绑定操作引入。程序文本中名称的每一次出现都会引用名称的绑定，这种绑定在包含名称使用的最内层函数块中建立。

块是Python 程序文本的一个片段，作为一个单元执行。下面这些都是块：模块、函数体、类定义。交互式敲入的每一个命令都是块。一个脚本文件（作为解释器标准输入的文件或者在解释器的命令行中指定的第一个参数）是一个代码块。一个脚本命令（在解释器的命令行中以'**-c**'选项指定的命令）是一个代码块。由内建函数execfile()读入的文件是一个代码块。传递给eval()内建函数和exec语句的字符串参数是一个代码块。由内建函数input()读入并执行的表达式是一个代码块。

代码块在执行帧中执行。帧包含一些管理信息（用于调试）并决定代码块执行完成之后从哪里以及如何继续执行。

定义域定义块中名称的可见性。如果局部变量定义在一个块中，它的定义域就包含这个块。如果定义出现在函数块中，定义域将扩展到该定义块中所包含的任何块，除非被包含的块为该名称引入一个不同的绑定。类代码块中定义的名称的定义域限制在类代码块中；它不会扩展到方法的代码块中 - 包括生成器表达式，因为它们使用函数的定义域实现。这意味着下面的写法将会失败：

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

当名称在代码块中使用时，使用包含它的最内层定义域解析。对于一个代码块可见的定义域集合称为该代码块的环境。

如果名称绑定在代码块中，那么它是该代码块的局部变量。如果名称绑定在模块的级别，那么它是一个全局变量。（模块代码块的变量既是局部的又是全局的。）如果一个变量在代码块中使用但是没有在那里定义，那么它是自由变量。

当完全找不到名称时，引发一个NameError异常。如果名称引用一个没有绑定的局部变量，引发一个UnboundLocalError异常。UnboundLocalError是NameError的子类。

下面的句法结构将会绑定名称：函数的形式参数、import语句、类和函数定义（绑定类或函数的名称于定义它们的代码块中）、出现在赋值语句中的目标是标识符、for循环的头部、except子句头部的第二个位置或者with语句中as的后面。from...import*形式的import语句绑定导入的模块中定义的所有名称，除了以下划线开头的那些。这种形式只可以在模块级别使用。

出现在del语句中的目标也被认为是这种目的的绑定（尽管真实的语义是取消名称的绑定）。解绑定一个被封闭的定义域引用的名称是非法的；编译器将会报一个SyntaxError。

每个赋值和导入语句出现在类或函数定义的代码块中或者出现在模块级别（顶级代码块）。

如果在代码块的任意地方出现名称绑定操作，那么代码块中该名称的所有使用将被当做对当前代码块的引用。当名称在一个代码块中绑定之前使用时将导致错误。这个规则是微妙的。Python缺少声明并允许名称绑定操作出现在代码块内任何地方。代码块的局部变量通过扫描代码块的全部文本的名称绑定操作决定。

如果代码块中出现global语句，那么所有使用该语句指明的名称都是引用顶级命名空间中的名称绑定。顶级命名空间中名称通过查找全局命名空间解析，即模块的命名空间包括代码块、内建的命名空间以及**builtin**模块的命名空间。首先查找全局命名空间。如果那里找不到名称，就会查找内建的命名空间。global语句必须在该名称的所有使用之前。

与执行的代码块相关联的内建命名空间实际上是通过查找它的全局命名空间中的**builtins**名称找到的；它应该是一个字典或者一个模块（如果是后一种情况，将使用模块的字典）。默认情况下，在**main**模块中时，**builtins**就是内建的模块**builtin**（注意：没有's'）；在其它任何模块的时候，**builtins**是**builtin**模块自身的字典的别名。**builtins**可以被设置成一个用户创建的字典以创建一个严格执行的弱形式。

CPython实现细节：使用者不应该触动**builtins**；严格地讲它是实现细节。使用者如果想要覆盖内建命名空间中的变量应该导入该**builtin**（没有's'）模块并适当地修改它的属性。

模块的命名空间在模块第一次导入时自动创建。脚本的主模块总是被称为**main**。

global语句具有和同一个代码块中名称绑定操作相同的定义域。如果包含自由变量的最内层定义域包含一条global语句，那么这个自由变量被认为是一个全局变量。

类定义是一条可以使用和定义名称的可执行语句。这些引用遵循名称解析的正常规则。类定义的命名空间变成类的属性字典。类定义域中定义的名称在方法中不可见。

4.1.1. 与动态功能的交互

当与包含自由变量的嵌套定义域联合使用的时候，有几种Python语句是非法的情况。

如果变量在一个包含它的定义域中被引用，那么删除它的名称是非法的。在编译的时刻将会报告一个错误。

如果在函数中使用通配符形式的导入— import* —并且函数包含或者是一个带有自由变量的嵌套代码块，那么编译器将会引发一个SyntaxError。

如果函数中使用exec且函数包含或者是一个带有自由变量的嵌套代码块，那么编译器将会引发一个SyntaxError除非执行为exec显式指明局部命名空间。（换句话说，execobj是非法的，而execobjinns是合法的。）

eval()、execfile()和input()函数以及exec语句没有访问完整环境的权限来解析名称。名称可以在调用者的局部和全局命名空间中解析。自由变量不是在包含它们的最内层命名空间中解析，而是在全局命名空间中。[1]exec语句以及eval()和execfile()函数具有可选参数以覆盖全局和局部命名空间。如果只指明一个命名空间，则两个命名空间都会使用它。

4.2. 异常

异常是一种打断代码块的正常控制流程以处理错误或者其它意外情况的方法。异常在错误检测到的点引发；它可以通过包围它的代码块或者直接或间接调用发生错误的代码块的代码块处理。

Python解释器在检测到运行时错误（例如除0）时会引发一个异常。Python还可以通过raise语句显示地引发异常。异常处理器通过try ... except语句指定。这种语句的finally子句可以用来指定清除代码，它不处理异常，而是在前面的代码中无论有没有出现异常都会执行。

Python 异常处理使用“终止”模型：异常处理器可以查明发生了什么并在外层继续执行，但是它不可以修复错误的根源并重试失败的操作。（除非通过从顶层重新进入出错的代码片段）。

当一个异常没有被任何处理，那么解释器会终止程序的执行或者返回到其交互式的主循环。在任何一种情况下，它都会打印出一个栈回溯，除了异常是SystemExit的时候。

异常通过类的实例标识。except子句的选择依赖于类的实例：它必须引用实例的类或者其基类。实例可以通过处理器接收并且可以带有异常条件的额外信息。

异常也可以通过字符串标识，在这种情况下except子句通过对象的ID 选择。任意一个值可能会跟随标识字符串一起引发并传递给处理器。

注意

异常的消息不是Python API的一部分。它们的内容可能随着Python 版本不断地改变而没有警告，在多种不同版本的解释器下运行的代码不应该依赖这些内容。

另请参考*try语句*小节中的try语句和*raise语句*小节中的raise语句。

脚注

| [1] | 出现这种限制是因为通过这些操作执行的代码在模块编译的时候不可以访问。 | |-----|-----|

5. 表达式

这一章解释Python中表达式的各个组成部分的含义。

关于语法：在这一章及随后的章节中所用到的扩展BNF语法符号将用于讲述语法，而不是词法分析。当一个语法规则具有这样的形式

```
name ::=  othername
```

且没有给出语义，那么这种形式的name语义与othername相同。

5.1. 算术转换

当下面算术操作符的描述使用短语“数字参数被转换为一个共同的类型”时，这些参数将使用隐式转换规则列出的规则做隐式转换。如果两个参数都是标准的数字类型，那么运用下面的隐式转换：

- 如果任意一个参数是复数，将另外一个转换成复数；
- 否则，如果任意一个参数是浮点数，则将另外一个转换为浮点数；
- 否则，如果任意一个是长整数，则将另外一个转换成长整数；
- 否则，两个参数必定都是普通的整数且不必要转换。

某些特定的操作符适用其它的一些规则（例如，‘%’操作符左边的字符串参数）。解释器的扩展可以定义它们自己的转换规则。

5.2. 原子

原子是表达式最基础的元素。最简单的原子是标识符和字面值。在引号、圆括号、方括号或者花括号中的封闭形式在语法上也被分类为原子。原子的语法为：

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display
               | generator_expression | dict_display | set_display
               | string_conversion | yield_atom
```

5.2.1. 标识符（名称）

一个以原子出现的标识符是一个名称。词法定义请参看标识符和关键字小节，名称和绑定的文档请参看名称和绑定小节。

当名称绑定到一个对象上时，对该原子的求值将产生那个对象。当名称没有绑定时，试图对它求值将抛出NameError异常。

私有变量名称的改编：当出现在类定义中的标识符以两个或多个下划线字符开始且不是以两个或多个下划线结束，它被认为是那个类的私有名称。在为它们生成代码之前，私有名称被转换为更长的形式。该转换在名称在前面插入类的名称，前导的下滑线被删除并插入一个单一的下划线。例如，出现在类Ham中的标识符**spam**将被转换成**_Ham**spam。这种转换与标识符使用的语法上下文无关。如果转换后的名称过长（超过255个字符），将可能发生与具体实现有关的截断。如果类的名称只由下划线组成，则不会转换。

5.2.2. 字面值

Python支持字符串字面值和各种数值字面值：

```
literal ::=  stringliteral | integer | longinteger
             | floatnumber | imagnumber
```

在浮点数和虚数（复数）情况下，可能只是近似值。详细信息参见字面值一节。

所有的字面值都是不可变数据类型，因此对象的ID不如它的值重要。多次计算具有相同值的字面值（无论是程序文本中相同的出现还是不同的出现）得到的既可能是同一个对象也可能是具有相同值的不同对象。

5.2.3. 圆括号式

圆括号式是包含在圆括号中的一个可选表达式序列：

```
parenth_form ::=  "(" [expression_list] ")"
```

圆括号中的表达式序列产生的就是该表达式序列产生的内容：如果序列包含至少一个逗号，那么它产生一个元组；否则，它产生组成表达式序列的那个单一表达式。

空的圆括号对产生空的元组对象；因为元组是不可变的，字面值的规则同样适用（例如空元组的两次出现可能产生相同或不同的对象）。

注意元组不是通过圆括号而是逗号操作符形成。有个例外是空元组，它必须要有圆括号 — 允许表达式中出现没有括号的“空白”将导致歧义并使得常见的拼写错误无法发现。

5.2.4. 列表表示式

列表表示式是在方括号中的可以为空的一系列表达式：

```
list_display        ::=  "[" [expression_list | list_comprehension] "]"
list_comprehension  ::=  expression list_for
list_for            ::=  "for" target_list "in" old_expression_list [list_iter]
old_expression_list ::=  old_expression [("," old_expression)+ [","]]
old_expression      ::=  or_test | old_lambda_expr
list_iter           ::=  list_for | list_if
list_if             ::=  "if" old_expression [list_iter]
```

列表的表示式产生一个新的列表对象。它的内容通过提供一个表达式序列或者一个列表推导式指定。当提供的是一个逗号分隔的表达式序列时，对它的元素从左向右求值并按此顺序放入列表对象中。当提供的是一个列表推导式时，它由一个单一的表达式后面跟着至少一个for子句和零个或多个for或者if子句组成。在这种情况下，新的列表的元素是由for或者if子句块产生，这些子句块从左向右嵌套，且当到达最内层的代码块时对表达式求值以产生一个列表元素[1]。

5.2.5. 集合和字典的表示式

对于构造集合和字典，Python提供特殊的语法叫做“表示式"，它们有两种方式：

- 容器的内容被显式地列出，或者
- 它们通过一系列循环和过滤指令计算得到，这种方式叫做推导式。

推导式常见的语法元素为：

```
comprehension ::=  expression comp_for
comp_for      ::=  "for" target_list "in" or_test [comp_iter]
comp_iter     ::=  comp_for | comp_if
comp_if       ::=  "if" expression_nocond [comp_iter]
```

推导式由一个单一的表达式后面跟随至少一个for子句加上零个或多个for或者if子句。在这种情况下，新的列表的元素是由for或者if子句块产生，这些子句块从左向右嵌套，且当到达最内层的代码块时对表达式求值以产生一个列表元素。

注意，推导式在一个单独的作用域中执行，所以在目标序列中赋值的名称不会“泄露”到外围的作用域中。

5.2.6. 生成器表达式

生成器表达式是在圆括号中的一个简洁的生成器符号：

```
generator_expression ::=  "(" expression comp_for ")"
```

生成器表达式产生一个新的生成器对象。它的语法与推导式相同，只是它位于圆括号而不是方括号或花括号中。

生成器表达式中使用的变量在为生成器对象调用**next**()方法时才会惰性地求值（与普通的生成器方式相同）。但是，最左边的for子句会立即计算，所以它产生的错误可以在生成器表达式代码中的任何其它可能的错误之前发现。随后的for子句不可以立即计算因为它们可能依赖于前面的for循环。例如：(x*yforxinrange(10)foryinbar(x))。

圆括号对于只有一个参数的调用可以省略。细节请参考调用一节。

5.2.7. 字典表示式

字典表示式是在花括号中的可以为空的一系列键/值对。

```
dict_display       ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list     ::=  key_datum ("," key_datum)* [","]
key_datum          ::=  expression ":" expression
dict_comprehension ::=  expression ":" expression comp_for
```

字典表示式产生一个新的字典对象。

如果给出逗号分隔的键/值对序列，将从左向右对它们求值以定义字典中项：用每个键对象作为字典的键并存储对应的值。这意味着你可以在键/值序列中多次指定相同的键，但是该键最终对应的字典的值将是最后给出的那个值。

字典推导式，与列表和集合推导式相比，需要两个冒号分隔的表达式并在后面跟随通常的“for”和“if”子句。当推导执行时，产生的键和值以它们生成的顺序插入到新的字典中。

键的类型的限制在前面的标准类型的层次一节中有列出。（简要地讲，键的类型应该是可哈希的，即排除所有可变的对象。）重复的键之间的冲突不会被检测到；一个给定的键的最后的值（表示式中最右边的值）将获胜。

5.2.8. 集合表示式

集合表示式通过花括号表明，与字典表示式的区别是缺少冒号分隔的键和值：

```
set_display ::=  "{" (expression_list | comprehension) "}"
```

集合表示式产生一个新的可变集合对象，它的内容可以通过一个表达式序列或者一个推导式指定。当提供的是一个逗号分隔的表达式序列时，将从左向右计算它的元素并添加到集合对象中。当提供的是一个推导式时，集合根据推导式产生的元素构造。

不可以用{}构造一个空集合；该字面值构造一个空的字典。

5.2.9. 字符串转换式

字符串转换式是包含在反引号中的一个表达式序列：

```
string_conversion ::=  "`" expression_list "`"
```

字符串转换式计算包含的表达式序列并根据结果对象类型的特定规则将结果对象转换成一个字符串。

如果对象是一个字符串、一个数字、None或者一个只包含这些类型的对象，那么结果字符串将是一个合法的Python表达式，它可以传递给内建的eval()函数以产生一个具有相同值的表达式（或者近似值，如果调用的是浮点数）。

（特别地，字符串转换式会添加引号并将“古怪”的字符转换为转义的序列，这些序列打印出来是安全的。）

递归的对象（例如，直接或间接包含自身引用的列表或字典）使用...来表示一个递归的引用，其结果不可以传递给eval()以获得一个相等的值（将引发SyntaxError）。

内建函数repr()对其参数所做的转换与将它放入圆括号和反引号中完全相同。内建函数str()完成类似但更友好的转换。

5.2.10. Yield 表达式

```
yield_atom       ::=  "(" yield_expression ")"
yield_expression ::=  "yield" [expression_list]
```

2.5 版中新增。

yield表达式只用于定义生成器函数，且只能用于函数的定义体中。在函数定义中使用yield表达式就可以充分使得该定义创建一个生成器函数而不是普通的函数。

当调用生成器函数时，它返回一个称为生成器的迭代器。然后该生成器控制生成器函数的执行。当调用生成器的其中一个方法时，执行开始。此时，执行会行进到第一个yield表达式，在那里执行被挂起并返回expression_list的值给生成器的调用者。挂起的意思是保存所有的局部状态，包括当前局部变量的绑定、指令的指针和内部的计算栈。当通过调用生成器的一个方法来恢复执行时，函数可以准确地继续执行就好像yield表达式只是一个外部的调用。恢复执行后yield表达式的值取决于恢复执行的方法。

所有这些使得生成器函数与协程非常类似；它们可以yield多次，它们有多个入口点且它们的执行可以挂起。唯一的区别是生成器函数不可以控制yield之后执行应该从何处继续；控制始终被转让给生成器的调用者。

5.2.10.1. 生成器迭代器的方法

该小节讲述生成器迭代器的方法。它们可用于控制生成器函数的执行。

注意当生成器已经在执行时调用下面的任何一个生成器方法都将引发ValueError异常。

*class*generatorgenerator.next() 开始生成器函数的执行或者在最后一次执行的yield表达式处恢复执行。当生成器函数使用next()方法恢复执行时，当前的yield表达式始终None。然后执行继续行进到下一个yield表达式，在那里生成器被再次挂起并返回expression_list的值给next()的调用者。如果生成器退出时没有yield另外一个值，则引发一个StopIteration异常。

.generator.send(*value*) 恢复执行并"发送"一个值到生成器中。该value参数成为当前yield表达式的结果。send()方法返回生成器yield的下一个值，如果生成器退出时没有yield另外一个值则引发StopIteration。当调用send()用于开始生成器的执行时，它必须以None作为参数进行调用，因为没有接受该值的yield表达式。

generator.throw(*type*[, *value*[, *traceback*]]) 在生成器暂停的地方引发一个type类型的异常，并返回生成器函数yield的下一个值。如果生成器在退出时没有yield一个值，则引发StopIteration异常。如果生成器函数没有捕获传递进来的异常或者引发一个不同的异常，那么该异常将传播到调用者。

generator.close() 在生成器函数暂停的地方引发一个GeneratorExit。如果生成器函数此后引发StopIteration（正常退出或者由于已经正在关闭）或者GeneratorExit（没有捕获该异常），close会返回到调用者。如果生成器yield一个值，则引发一个RuntimeError。如果生成器引发其它任何异常，它会被传播到调用者。如果生成器已经由于异常退出或正常退出，close()不会做任何事情。

这里有个简单的例子演示生成器和生成器函数的行为：

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

另请参阅

PEP 0342 - 通过增强的生成器实现协程增强生成器API和语法的提议，使得它们可以作为简单的协程使用。

5.3. 初级操作

初级操作表示语言中绑定性最高的操作。它们的语法是：

```
primary ::=  atom | attributeref | subscription | slicing | call
```

5.3.1. 属性引用

属性引用是一个初级操作，后面跟随一个句号和一个名称：

```
attributeref ::=  primary "." identifier
```

primary必须是一个支持属性引用类型的对象，例如模块、列表和实例。接着该对象被要求生成名称为identifier的属性。如果该属性不可访问，则抛出AttributeError异常。否则，生成的对象的类型和值取决于该对象。对相同属性的多次求值可能产生不同的对象。

5.3.2. 下标

下标选择序列（字符串、元组或列表）或者映射（字典）对象的一个元素：

```
subscription ::=  primary "[" expression_list "]"
```

primary必须是一个序列或者映射类型的对象。

如果primary是一个映射，那么expression_list必须是一个对象，其值为映射的一个键，该下标选择映射中对应于该键的值。（expression_list是一个元组除非它只有一个元素。）

如果primary是一个序列，那么expression_list必须是一个普通的整数。如果该值是负数，则加上该序列的长度（所以，x[-1]选择x的最后一个元素。）结果值必须是一个小于序列元素个数的非负整数，下标操作选择索引为该值的元素（从零开始计数）。

字符串的元素为字符。字符不是一个单独的数据类型而是只有一个字符的字符串。

5.3.3. 切片

切片选择序列对象（例如，字符串、元组和列表）中一个范围内的元素。切片可以用作表达式或者作为赋值和del语句的目标。切片的语法：

```
slicing          ::=  simple_slicing | extended_slicing
simple_slicing   ::=  primary "[" short_slice "]"
extended_slicing ::=  primary "[" slice_list "]"
slice_list       ::=  slice_item ("," slice_item)* [","]
slice_item       ::=  expression | proper_slice | ellipsis
proper_slice     ::=  short_slice | long_slice
short_slice      ::=  [lower_bound] ":" [upper_bound]
long_slice       ::=  short_slice ":" [stride]
lower_bound      ::=  expression
upper_bound      ::=  expression
stride           ::=  expression
ellipsis         ::=  "..."
```

这里的形式语法有歧义：expession_list看上去也像slice_list，所以任何下标也可以解释为切片。为了不引入更复杂的语法，通过定义在这种情况下解释为下标优先于解释为切片来消除歧义（如果slice_list不包含proper_slice和ellipse也属于这种情况）。类似地，当slice_list只有一个short_slice且没有末尾的逗号时，解释为简单切片要优先于解释为扩展切片。

简单切片的语义如下。primary必须是一个序列对象。下界和上界表达式，如果存在，必须是普通的整数；默认分别是零和sys.maxint。如果有一个是负数，则将它加上序列的长度。切片选择所有索引为*k*的元素，其中i<=k<j且*i*和*j*是指定的下界和上界。它可能是一个空的序列。如果*i*或者*j*位于合法的索引范围之外不会出错（这些元素不存在所以它们不会被选择）。

扩展切片的语法如下。primary必须是一个映射对象，它以从slice_list构造的键做索引，如下所示。如果slice_list包含至少一个逗号，则键是一个包含slice_item转换的元组；否则，long_slice作为键。slice_item是一个表达式时，转换就是那个表达式。slice_item是ellipsis时，转换为内建的Ellipsis对象。proper_slice的转换是一个切片对象（参阅标准类型的层次），它的start、stop和step属性分别是表达式lower_bound、upper_bound和stride的值，没有的表达式用None代替。

5.3.4. 调用

调用是指用一个可以为空的参数序列调用一个可调用对象（例如，一个函数）：

```
call                 ::=  primary "(" [argument_list [","]
                          | expression genexpr_for] ")"
argument_list        ::=  positional_arguments ["," keyword_arguments]
                            ["," "*" expression] ["," keyword_arguments]
                            ["," "**" expression]
                          | keyword_arguments ["," "*" expression]
                            ["," "**" expression]
                          | "*" expression ["," "*" expression] ["," "**" expression]
                          | "**" expression
positional_arguments ::=  expression ("," expression)*
keyword_arguments    ::=  keyword_item ("," keyword_item)*
keyword_item         ::=  identifier "=" expression
```

在位置参数和关键字参数之后可以存在一个末尾的逗号而不影响语义。

primary必须是一个可调用的对象（用户定义的函数、内建的函数、内建对象的方法、类对象、类实例的方法以及某些类实例自己也是可调用的；Python的扩展可以定义额外的可调用对象类型）。所有的参数表达式都将在调用发生之前求值。关于形式参数列表的语法请参考函数的定义一节。

如果存在关键字参数，它们首先被转换为位置参数，如下所述。首先，创建一个没有填充的空位序列用于形参。如果有N个位置参数，则它们被放置在前N个空位中。下一步，对于每个关键字参数，用标识符决定对应的位置（如果标识符与第一个形参的名称相同，则使用第一个位置，以此类推）。如果该位置已经被填充，则引发一个TypeError异常。否则，将该参数的值放入该位置并填充它（即使该表达式是None，它也会填充该位置）。当处理完所有的参数时，仍然没有填充的位置将用来自函数定义的对应默认值填充。（默认值只在函数定义时计算一次；因此，用于默认值的可变对象例如列表或字典将被所有没有指定对应位置参数值的调用共享；通常应该避免这点。）如果有没有填充的空位且没有指定默认值，则引发一个TypeError异常。否则，使用这些填满的位置作为调用的参数序列。

CPython实现细节：一种实现可以提供这样的内建函数，它的位置参数没有名称，因此不可以通过关键字提供，即使它们由于文档的需要而被“命名”。在CPython中，那些用C实现并使用PyArg_ParseTuple()解析参数的函数就是这种情况。

如果位置参数的个数多于形参，则引发一个TypeError异常，除非存在一个使用*identifier语法的形参；在这种情况下，该形参接收一个包含多余位置参数的元组（如果没有多余的位置参数则为空元组）。

如果有任何关键字参数没有对应的形参名称，则引发一个TypeError异常，除非存在一个使用**identifier语法的形参；在这种情况下，该形参接收一个包含多余的关键字参数的字典（使用关键字作为键，参数值作为对应的值），如果没有多余的关键字参数则为一个（新的）空字典。

如果*expression*语法出现在函数调用中，那么*expression*必须是一个可迭代器。来自该可迭代器的元素被当作额外的位置参数；如果位置参数为x1, ..., xN且*expression*求值为一个序列y1, ..., yM，那么它等同于用*M+N*个位置参数x1, ..., xN, y1, ..., yM*的调用。

这种方式的后果是虽然*expression*可以出现在某些关键字参数之后，但是它将在关键字参数（以及***expression*参数–见下文）之前*处理。所以：

```
>>> def f(a, b):
...  print a, b
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

关键字参数和*expression语法都在同一个调用中使用很不常见，所以实践中这种困惑不会出现。

如果**expression语法出现在函数调用中，那么expression必须是一个映射，它的内容将被当作额外的关键字参数。某个关键字在expression和显式的关键字参数中都出现的情况下，将引发TypeError异常。

使用*identifier*或者*identifier语法的形参不可以作为位置参数的槽位或者关键字参数的名称使用。使用(sublist)语法的形参不可以作为关键字参数的名称使用；最外层的sublist对应于单一的一个未命令参数槽位，在所有其它参数的处理完成之后参数的值使用普通的元组赋值规则赋值给sublist。

一个调用永远返回某个值，即使可能为None，除非它引发一个异常。该值如何计算取决于可调用对象的类型。

如果它是—

一个用户定义的函数：该函数的代码块将被执行，并将参数列表传递给它。代码块所做的第一件事是将绑实参与参数定；这在函数的定义一节有讲述。当代码块执行一条return时，它表示该函数调用的返回值。

一个内建的函数或方法：结果取决于解释器；内建函数和方法的描述请参见内建的函数 。

一个类对象：返回该类的一个新的实例。

一个类实例的方法：调用对应的用户自定义的函数，参数列表比调用的参数列表多一个元素：实例成为第一个参数。

一个类实例：该类必须定义一个**call()**方法；效果和调用该方法一样。

5.4. 乘方操作符

乘方操作符的绑定性比它左侧的一元操作符高；比它右侧的一元操作符绑定性低。其语法是：

```
power ::=  primary ["**" u_expr]
```

因此，在一个没有括号的乘方和一元操作符序列中，操作符从右向左计算（这不会约束操作数的计算顺序）：-1**2的结果是-1。

乘方操作符的语义与用两个参数调用内建的pow()函数相同：它产生左侧参数的右侧参数次幂。其数值参数首先被转换成相同的类型。结果的类型是强制转换后的参数类型。

操作符的类型不一样时，运用二元算术操作符的强制转换规则。对于整数和长整数，结果的类型与（强制转换后的）操作数类型相同，除非第二个参数为负数；在这种情况下，所有的参数被转换成浮点数并返回浮点数结果。例如，10**2**返回**100**，但是**10**-2返回0.01。（最后的这个特性在Python2.2中添加。在Python2.1和之前的版本中，如果两个参数都是整数且第二个参数为负，则会引发一个异常）。

0.0的负数乘方将导致ZeroDivisionError。负数的小数次幂将导致ValueError。

5.5. 一元算术和位操作

所有的一元算术和位操作具有相同的优先级：

```
u_expr ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

一元-（负）操作符产生其数值参数的负值。

一元+（正）操作符产生其数值参数保持不变。

一元~（反）操作符产生其普通整数和长整数参数按位取反的值。x按位取反定义为-(x+1)。它只适用于整数数值。

在这三种情形中，如果参数的类型不合适，都将引发TypeError异常。

5.6. 二元算术操作

二元算术操作具有传统的优先级。注意这里的某些操作同样适用于一些非数值类型。除了乘方操作符，有两个优先级，一个针对乘法操作符，一个针对加法操作符：

```
m_expr ::=  u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr
            | m_expr "%" u_expr
a_expr ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

*（乘法）操作符产生它参数的乘积。其参数必须都是数值，或者一个是整数（普通整数或长整数）另外一个是序列。在前一种情况下，数值会被转换成一个相同的类型然后一起相乘。在后一种情况下，将进行序列的重复操作；负的重复值将产生一个空的序列。

/（除法）和//（整除）操作符产生它们参数的商。其数值参数首先被转换成相同的类型。普通整数或者长整数除法产生一个相同类型的整数；其结果是在算术除法的结果上调用“取整”函数。除以零将引发ZeroDivisionError异常。

%（取模）操作符产生第一个参数除以第二个参数后的余数。其数值参数将首先被转换成相同的类型。右边的参数为零将引发ZeroDivisionError异常。参数可以是浮点数，例如，3.14%0.7等于0.34（因为3.14等于4*0.7+0.34。）取模操作符永远产生与第二个操作符符号相同的结果（或者为零）；结果的绝对值将严格小于第二操作数的绝对值[2]。

整数的除法和取模操作由下面的等式相关联：x==(x/y)*y+(x%y)。整数除法和取模同样由内建函数divmod()相关联：divmod(x,y)==(x/y,x%y)。这些等式对于浮点数不成立；类似的等式在x/y替换为floor(x/y)或者floor(x/y)-1时近似成立[3]。

除了执行整数的取模操作，%操作符还被字符串和unicode对象重载用于执行字符串的格式化（也叫做插值）。字符串格式化的语法在Python库参考的字符串格式化一节讲述。

2.3版后废弃的内容：整除操作符、取模操作符和divmod()函数不再为复数定义。作为替代，如果合适，使用将abs()函数将其转换为浮点数。

+（加法）操作符产生其参数的和。其参数必须都是数值或者都是相同类型的序列。在前一种情况下，数值被转换成相同的类型然后一起相加。在后一种情况下，序列被连接在一起。

-（减法）操作符产生其参数差。其参数首先被转换成相同的类型。

5.7. 移位操作

移位操作的优先级低于算术操作：

```
shift_expr ::=  a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

这些操作符接收普通整数或者长整数作为参数。参数会被转换成一种共同的类型。它们将第一个参数向左或向右移动第二个参数指出的位数。

右移*n*位定义为除以pow(2,n)。左移*n*位定义为乘以pow(2,n)。负的移位数目会引发ValueError异常。

注

在当前的实现中，要求右操作数至多为sys.maxsize。如果右操作数大于sys.maxsize，将引发OverflowError异常。

5.8. 二元位操作

下面三种位操作具有各自不同的优先级：

```
and_expr ::=  shift_expr | and_expr "&" shift_expr
xor_expr ::=  and_expr | xor_expr "^" and_expr
or_expr  ::=  xor_expr | or_expr "|" xor_expr
```

&操作符产生按位与，它的参数必须是普通整数或长整数。参数会被转换成一种共同的类型。

^操作符产生按位异或，它的参数必须是普通整数或长整数。参数会被转换成一种共同的类型。

|操作符产生按位或，它的参数必须是普通整数或长整数。参数会被转换成一种共同的类型。

5.9. 比较操作

与C不同，Python中所有的比较操作具有相同的优先级，并低于任何算术、移位和位操作。与C不同的还有，类似a<b<c 这样的表达式就是数学中传统的含义。

```
comparison    ::=  or_expr ( comp_operator or_expr )*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                   | "is" ["not"] | ["not"] "in"
```

比较操作产生两个值：True 或者False。

比较操作可以任意连接，例如x<y<=z等同于x<yandy<=z，但是y只计算一次（在两种情况中当发现x<y为假时都不会再计算z）。

形式上，如果*a*, *b*, *c*, ..., *y*, *z*是表达式且*op1*, *op2*, ..., *opN*是操作数，那么aop1bop2c...yopNz等同于aop1bandbop2cand...yopNz，不同点是每个表达式值至多计算一次。

注意aop1bop2c并不意味着*a* 和*c* 之间会有比较，所以xz 是完全合法的（尽管不漂亮）。

<>和!=两种形式时等同的；为了与C保持一致，倾向于使用!=；下面提到!=的地方同样接受<>。拼写<>被认为是废弃的。

操作符<、>、==、>=、<=和!= 比较两个对象的值。对象不需要具有相同的类型。如果两个都是数字，它们将被转会成一个共同的类型。否则，不同类型的对象将永远是不相等的，虽然顺序是固定的但是是随机的。你可以通过定义**cmp**方法或者像**gt**这样更丰富的比较方法来控制非内建类型对象的比较行为，在*Special method names*一节有详细的描述。

（这种不常见的比较定义用于简化像排序这种操作的定义以及in和notin操作符。未来，不同类型对象的比较规则可能会改变。）

相同类型对象的比较取决于它们的类型：

- 数字按照算术意义比较。
- 字符串使用字符对应的数值（内建函数ord()的结果）按字典序比较。 Unicode和八比特字符完全适用这种行为。[4]
- 元组和列表通过比较对应的项按字典序比较。这意味着若要比较结果相等，每一个元素比较的结果必须相等且两个序列的类型必须相同并且具有相同的长度。

如果不相等，序列按照它们第一个不同的元素排序。例如，cmp([1,2,x],[1,2,y])的返回值与cmp(x,y)相同。如果对应的元素不存在，则长度较短的序列排在第一个（例如[1,2]<[1,2,3]）。

- 映射（字典）当且仅当它们排好序的（键，值）列表相等时才相等。[5]虽然当不相等时的结果是固定的，但是具体是怎样的结果是没有定义的。[6]
- 其它大部分内建类型的对象是不相等的除非它们是相同的对象；一个对象是小于还是大于另外一个对象的抉择虽然是随机的但是在程序的一次执行中是一致的。

操作符in和notin用于测试成员资格。如果*x* 是*s* 的一个成员那么xins为真，否则为假。xnotins返回xins的否定式。成员资格测试传统用于序列上；如果序列包含一个元素与某对象相等则该对象是这个序列的成员。然而，其它许多类型的对象不用称为序列而支持成员资格测试也是合理的。特别地，字典（针对键）和集合就支持成员关系测试。

对于列表和元组类型，xiny 为真当且仅当存在一个索引*i* 使得x==y[i]为真。

对于Unicode和字符串类型，xiny为真当且仅当*x* 是*y* 的一个子串。

版本2.3 中的改变：之前，*x*要求是一个长度为1的字符串。

对于定义了**contains()**方法的用户自定义类，xiny为真当且仅当y.**contains**(x)为真。

对于没有定义**contains()**但定义**iter()**的用户自定义类，xiny为真如果某个值z在迭代y时满足x==z。如果迭代过程中抛出异常，就好像是in抛出那个异常一样。

最后，尝试旧式的迭代协议：如果一个类定义了**getitem()**，xiny为真当且仅当有一个非负的整数索引*i*使得x==y[i], 且更小的索引不会引发IndexError异常。（如果引发了其它异常，则像是in引发了该异常）。

notin操作符定义为取与in相反的真值。

is 和isnot操作测试对象的ID：xisy当且仅当*x*和*y*是相同的对象时为真。xisnoty产生相反的真值。[7]

5.10. 布尔操作

```
or_test  ::=  and_test | or_test "or" and_test
and_test ::=  not_test | and_test "and" not_test
not_test ::=  comparison | "not" not_test
```

在布尔操作的上下文中，同时在控制流语句使用表达式的时候，以下的值被解释为假：False、None、所有类型中的数值零、空的字符串和容器（包括字符串、元组、字典和固定集合）。其它所有的值都解释为真。（请参见**nonzero()** 特殊方法以获得改变这种行为的方式。）

如果操作符not的参数为假则其产生True，否则产生False。

表达式xandy首先计算*x*；如果*x*为假，则返回它的值；否则，再计算*y* 并返回结果的值。

表达式xory首先计算*x*；如果*x*为真，则返回它的值；否则，再计算*y* 并返回结果值。

（注意and和or的返回值都不局限于False和True，而是返回最后计算的参数结果。这在有些时候是有用的，例如，如果s是一个字符串，当它是空的时候应该被一个默认值替换，那么表达式sor'foo'就可以产生想要的值。因为not无论如何必须生成一个值，它不会设法返回和其参数类型相同的值，所以not'foo'生成False, 而非''。）

5.11. 条件表达式

在版本2.5 中新引入。

```
conditional_expression ::=  or_test ["if" or_test "else" expression]
expression             ::=  conditional_expression | lambda_expr
```

条件表达式（有时叫做“三元操作符”）在所有的Python操作符中具有最低的优先级。

表达式xifCelsey首先计算条件*C*（而非*not***x*）；如果*C*为真，则计算*x* 并返回它的值；否则，计算*y* 并返回它的值。

关于条件表达式的更多细节，请参见**PEP 308**。

5.12. Lambda 表达式

```
lambda_expr     ::=  "lambda" [parameter_list]: expression
old_lambda_expr ::=  "lambda" [parameter_list]: old_expression
```

Lambda表达式（有时叫做lambda形式）具有和表达式相同的地位。它们是创建匿名函数的一种快捷方式；表达式lambdaarguments:expression生成一个函数对象。此命名对象的行为类似下面定义的函数对象

```
def name(arguments):
    return expression
```

关于参数列表的语法，请参见函数定义一节。注意lambda表达式创建的函数不可以包含语句。

5.13. 表达式序列

```
expression_list ::=  expression ( "," expression )* [","]
```

至少包含一个逗号的表达式序列产生一个元组。元组的长度是列表中表达式的个数。表达式按从左到右的顺序计算。

尾部的逗号仅仅在创建单元素元组（又叫独元）时需要；在其它所有情况下，它都是可选的。

6. 简单语句

简单语句包含在单一的一个逻辑行中。几个简单语句可以用分号分隔出现在单一的一行中。简单语句的语法是：

```
simple_stmt ::=  expression_stmt
                 | assert_stmt
                 | assignment_stmt
                 | augmented_assignment_stmt
                 | pass_stmt
                 | del_stmt
                 | print_stmt
                 | return_stmt
                 | yield_stmt
                 | raise_stmt
                 | break_stmt
                 | continue_stmt
                 | import_stmt
                 | global_stmt
                 | exec_stmt
```

6.1. 表达式语句

表达式语句用于（主要用于交互式地）计算和写入一个值，或者（通常）调用一个过程（返回结果没有意义的函数；在Python中，过程返回None值）。其它表达式语句的使用也是允许的，但是很少有意义。表达式语句的语法是：

```
expression_stmt ::=  expression_list
```

表达式语句计算表达式列表的值（也可能是一个单一的表达式）。

在交互模式下，如果值不是None，它将被内建的repr()函数转换为一个字符串，产生的字符串被写到标准输出（参见*print*语句）的一行上。（生成None 的表达式不会被输出，因此过程调用不会带来任何输出。）

6.2. 赋值语句

赋值语句用于（重新）绑定名称到具体的值以及修改可变对象的属性或元素：

```
assignment_stmt ::=  (target_list "=")+ (expression_list | yield_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" target_list ")"
                     | "[" target_list "]"
                     | attributeref
                     | subscription
                     | slicing
```

（最后三种符号的定义参见初级操作一节。）

赋值语句计算expression_list（记住，它可以是一个单一的表达式也可以是一个逗号分隔的序列，后者生成一个元组）并且从左到右把单一的结果对象赋值给target_list的每一个元素。

赋值是递归定义的，取决于目标（序列）的形式。当目标是可变对象的一部分时（属性引用，下标或者切片），最终必须由该可变对象做赋值操作并决定其合法性，如果赋值不可接受可以抛出一个异常。各种类型遵守的规则以及抛出的异常根据对象类型的定义给出（参见标准类型的层次一节）。

赋值一个对象给一个目标序列按如下方式递归定义：

- 如果对象列表是单一的目标：对象赋值给该目标。
- 如果目标序列是逗号分隔的序列：对象必须是可迭代的且元素个数与目标序列中目标个数相同，然后元素从左向右赋值给对应的目标。

赋值一个对象给一个单一的目标按如下方式递归定义。

- 如果目标是一个标识符（名称）：

- 如果名称没有出现在当前代码块的global语句中：名称绑定到当前局部命名空间中的对象。

- 否则：名称绑定到当前全局命名空间的对象。

如果名称已经绑定，那么它将重新绑定。这可能导致之前绑定到该名称的对象的引用计数变为零，引起该对象被释放并调用它的析构函数（如果有的话）。

- 如果目标是一个包含在圆括号或者方括号中的目标序列：对象必须是可迭代的且元素个数与目标序列中目标个数相同，然后元素从左向右赋值给对应的目标。

- 如果目标是属性引用：计算引用中的初级表达式。它产生的对象应该具有一个可以赋值的属性；如果情况不是这样，则抛出TypeError异常。然后要求该对象将被赋值的对象赋值给给定的属性；如果不能做此操作，它会抛出一个异常（通常是AttributeError，但不一定）。

注意：如果对象是类的实例且属性引用出现在赋值运算符的两侧，那么右侧的表达式a.x既可以访问实例属性（如果不存在实例属性）也可以访问类属性。左侧的目标将a.x永远设置成实例的属性，如果必要将创建它。因此，a.x的两次出现不是一定会引用同一个属性：如果右侧表达式引用的是一个类属性，左侧的表达式将创建一个新的实例属性作为赋值的目标。

```
class Cls:
    x = 3             # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

这里的描述不一定适用描述器属性，例如property()创建的属性。

- 如果目标是下标操作符：计算引用中的初级表达式。它应该生成一个可变的序列对象（例如列表）或者映射对象（例如字典）。然后，计算下标表达式。

如果primary是一个可变的序列对象（例如一个列表），则下标必须产生一个普通的整数。如果它是负数，将会加上序列的长度。结果值必须是一个小于序列长度的非负整数，然后将要求序列赋值该对象给具有那个索引的元素。如果索引超出范围，则引发IndexError异常（给序列下标赋值不能添加新的元素到序列中）。

如果primary 是一个映射对象（例如一个字典），下标必须具有和映射的关键字类型兼容的类型，然后要求映射创建一个键/值对将下标映射到赋值的对象。这既可以替换一个具有相同键值得已存在键/值对，也可以插入一个新的键/值对（如果不存在相同的键）。

- 如果目标是一个切片：计算引用中的初级表达式。它应该产生一个可变序列对象（例如列表）。被赋值的对象应该是相同类型的序列对象。下一步，如果存在，则计算下边界和上边界表达式；默认是零和序列的长度。边界计算的值应该是（小）整数。如果任意一个边界为复数，则会给它加上序列的长度。结果求得的边界在零和序列的长度之间，包括边界在内。最后，要求序列对象用赋值的序列元素替换切片。切片的长度可能不同于赋值的序列的长度，因此如果对象允许则改变目标序列的长度。

CPython实现细节：在目前的实现中，目标的语法和表达式的语法相同，不合法的语法将在代码生成阶段被排除，导致不够详细的错误信息。

警告：虽然赋值的定义暗示左侧和右侧之间的交叉赋值是‘安全的’（例如，a,b=b,a交换两个变量），但是赋值目标集的内部有交叉则是不安全的！例如，下面的程序将打印[0,2]：

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.2.1. 增强的赋值语句

增强的赋值是将二元操作和赋值语句组合成一个单一的语句。

```
augmented_assignment_stmt ::=  augtarget augop (expression_list | yield_expression)
augtarget                 ::=  identifier | attributeref | subscription | slicing
augop                     ::=  "+=" | "-=" | "*=" | "/=" | "//=" | "%=" | "**="
                               | ">>=" | "<<=" | "&=" | "^=" | "|="
```

（最后三个符号的语法定义参见初级操作一节。）

增强的赋值将先求值target（和普通的赋值语句不同，它不可以是一个可拆分的对象）和expression_list，然后完成针对两个操作数的二元操作，最后将结果赋值给初始的target。target只计算一次。

像x+=1这样增强的赋值表达式可以重写成x=x+1以达到类似但不完全等同的效果。在增强版本中，x只计算一次。还有，如果可能，真实的操作是原地的，意思是不创建一个新的对象并赋值给target，而是直接修改旧的对象。

除了不可以在一个语句中赋值给元组和多个目标，增强的赋值语句完成的赋值和普通的赋值以相同的方式处理。类似地，除了可能出现的原地行为，增强的赋值完成的二元操作和普通的二元操作相同。

如果target是属性引用，关于类和实例属性的注意事项同样适用于正常的赋值。

6.3. assert 语句

Assert语句是插入调试断言到程序中的一种便捷方法：

```
assert_stmt ::=  "assert" expression ["," expression]
```

其简单形式，assertexpression，等同于

```
if __debug__:
   if not expression: raise AssertionError
```

其扩展形式，assertexpression1,expression2，等同于

```
if __debug__:
   if not expression1: raise AssertionError(expression2)
```

这些等价的语句假定**debug**和AssertionError引用的是同名的内建变量。在当前的实现中，内建的变量**debug**在正常情况下为为True，在要求优化时（命令行选项 -o）为False。在编译时刻，当要求优化时，目前的代码生成器不会为断言语句生成任何代码。注：不必把失败的表达式的源代码包含进错误信息；它将作为栈回溯的一部分显示出来。

给debug赋值是非法的。内建变量的值在解释器启动的时候就已决定。

6.4. pass 语句

```
pass_stmt ::=  "pass"
```

pass是一个空操作 — 执行它的时候，什么都没有发生。它的用处是当语法上要求有一条语句但是不需要执行任何代码的时候作为占位符，例如：

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

6.5. del 语句

```
del_stmt ::=  "del" target_list
```

删除是递归定义的，和赋值的定义方式非常相似。这里就不详细讲述完整的细节，只给出一些注意事项。

删除目标将从左向右递归删除每一个目标。

删除一个名称将从局部或全局命名空间中删除该名称的绑定，取决于名称是否出现在相同代码块的global语句中。如果名称没有绑定，将抛出一个NameError 异常。

如果名称作为自由变量出现在嵌套的代码块中，从局部命名空间中删除它是非法的。

属性引用、下标和切片的删除将传递给原始的对象；切片的删除在一般情况下等同于赋予一个右边类型的空切片（但即使这点也是由切片的对象决定）。

6.6. print 语句

```
print_stmt ::=  "print" ([expression ("," expression)* [","]]
                | ">>" expression [("," expression)+ [","]])
```

print依次计算每一个表达式并将求得的对象写入标准输出（参见下文）。如果对象不是字符串，那么首先使用字符串转换规则将它转换成字符串。然后输出（求得的或者原始的）字符串。在（转换和）输出每个对象之前会输出一个空格，除非输出系统认为它位于一行的开始。这些情况包括（1）还没有字符写入到标准输出，（2）写入到标准输出的最后一个字符是除"以外的空白字符，或者（3）最后向标准输出写入的操作不是print语句。（在某些情况下由于这个原因向标准输出写入一个空白字符串可能是有用的。）

注意

行为像文件对象但是不是内建的文件对象的对象通常不会恰当地模拟文件对象的这方面行为，所以最好不要依赖这个行为。

在结尾会写入一个'\n'字符，除非print语句以逗号结束。如果语句只包含关键字print，这将是唯一的行为。

标准输出定义为内建模块sys中名为stdout的对象。如果不存在该对象，或者它没有write()方法，将抛出一个RuntimeError异常。

print同样有一种扩展的形式，由上面描述的语法的第二部分定义。这种形式有时被称为“print chevron。”在这种形式中，>>之后的第一个表达式必须是一个“类文件”对象，具体点就是具有上面提到的write()方法的对象。通过这种扩展形式，随后的表达式被输入到该文件对象。如果第一个表达式求值为None，那么使用sys.stdout作为输出的文件。

6.7. return 语句

```
return_stmt ::=  "return" [expression_list]
```

return在语法上只可以出现在函数定义中，不可以出现在类定义中。

如果存在expression_list，则计算它，否则使用None替换。

return离开当前的函数调用时以expression_list（或None）作为返回值。

当return将控制传出带有finally子句的try语句时，在真正离开函数之前会执行finally子句。

在生成器函数中，return语句不允许包含expression_list。在这种情况下，空的return表明生成器已经完成并将导致StopIteration异常抛出。

6.8. yield 语句

```
yield_stmt ::=  yield_expression
```

yield语句只在定义生成器函数是使用，且只在生成器函数的函数体中使用。在函数定义中使用yield语句就足以创建一个生成器函数而不是普通函数。

当调用生成器函数时，它返回一个称为生成器迭代器的迭代器，或者通常就叫做生成器。生成器函数体的执行通过重复调用生成器的next()方法直到它抛出一个异常。

当执行一个yield语句时，生成器的状态将冻结起来并且将expression_list的值返回给next()的调用者。“冻结”的意思是所有局部的状态都会被保存起来，包括当前局部变量的绑定、指令指针、内部的计算栈：保存足够的信息以使得下次调用next()时，函数可以准确地继续，就像

yield语句只是另外一个外部调用。

从Python 2.5版开始，yield语句允许在出现在try ... finally结构的try子句中。如果生成器在终结（引用数达到零或者被当作垃圾回收）之前没有恢复，将调用生成器迭代器的close()方法， 这允许任何挂起的finally子句可以执行。

yield语义的完整细节，参考*Yield*表达式一节。

注意

在Python 2.2中，yield语句只有当generators功能启用了时才允许。**future**导入语句用来启用该功能：

```
from __future__ import generators
```

另请参阅

PEP 0255 - 简单的生成器添加生成器和yield语句到Python中的提议。**PEP 0342** - 通过增强的生成器实现协程该提议提出，除了其它的生成器的增强之外，允许yield出现在try ... finally代码块的内部。

6.9. raise 语句

```
raise_stmt ::=  "raise" [expression ["," expression ["," expression]]]
```

如果没有表达式，raise 重新抛出当前作用域中最后一个激活的异常。如果当前作用域中没有活着的异常，则抛出TypeError异常以表示这是一个错误（如果在IDLE中运行，则会抛出Queue.Empty异常）。

否则，raise计算后面的表达式以得到三个对象，使用None作为省略的表达式的值。前面的两个对象用于决定异常的类型和值。

如果第一个对象是一个实例，那么异常的类型是实例的类，实例本身是值，第二个对象必须是None。

如果第一个对象是一个类，那么它将成为异常的类型。第二个对象用于决定异常的值：如果它是类的实例，那么该实例将成为异常的值。如果第二个对象是一个元组，它用于类构造函数的参数列表；如果它是None，则使用一个空的参数列表，如果是其它任何对象则被当做构造函数的一个单一的参数。通过调用构造函数创建的实例将用作该异常的值。

如果存在第三个对象且不为None，那么它必须是一个回溯对象（参见标准类型的层次一节）， 且它将替换当前异常发生的位置。如果存在第三个对象且值不是回溯对象或者None，将会抛出TypeError 异常。具有三个表达式形式的raise用于在except子句中显式地重新抛出异

常，但是如果重新抛出的异常是当前作用域中最近激活的异常则应该优先使用不带表达式的raise。

关于异常的更多信息可以在异常一节中找到，如何处理异常的信息在*try*语句一节。

6.10. break 语句

```
break_stmt ::=  "break"
```

break在语法上只可以出现在for或者while循环中，但不能嵌套在这些循环内的函数和类定义中。

它终止最相近的循环，如果循环有else子句将跳过。

如果break终止了一个for循环，控制循环的目标保持当前的值。

当break将控制传出带有finally子句的try语句时，在离开循环之前会执行finally子句。

6.11. continue 语句

```
continue_stmt ::=  "continue"
```

continue在语法上只可以出现在for或while循环中，但不能嵌套在这些循环内的函数定义、类定义和finally子句中。它继续最内层循环的下一轮。

当continue将控制传出带有finally子句的try语句时，在真正开始下一轮循环之前会执行finally子句。

6.12. import 语句

```
import_stmt     ::=  "import" module ["as" name] ( "," module ["as" name] )*
                     | "from" relative_module "import" identifier ["as" name]
                     ( "," identifier ["as" name] )*
                     | "from" relative_module "import" "(" identifier ["as" name]
                     ( "," identifier ["as" name] )* [","] ")"
                     | "from" module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
name            ::=  identifier
```

import语句分两步执行：（1）找到模块，如果必要则进行初始化；（2）定义（import语句所在作用域的）局部命名空间中的名称。该语句有两种形式，区别在于有没有使用from关键字。第一种形式（没有from）针对序列中的每个标识符重复执行这些步骤。具有from的形式

将先执行一次步骤（1），然后重复执行步骤（2）。

为了理解步骤（1）如何发生，你必须首先理解Python如何处理模块的分层命名。为了帮助组织模块并提供一套命名的层级，Python有一个包的概念。包可以包含其它包和模块，但是模块不可以包含其它模块或者包。从文件系统的角度，包是目录而模块是文件。原始的包的说明仍然可以阅读，尽管自从该文档的编写以来小的细节已经发生了变化。

一旦知道模块的名字（除非特别指出，“模块”这个词兼指包和模块），模块或者包的搜索就可以开始。首先检查的地方是sys.modules，这里是之前已经导入的所有模块的缓存。如果找到该模块，那么将在导入的步骤（2）使用它。

如果在缓存中没有找到该模块，则搜索sys.meta_path（sys.meta_path的说明可以在**PEP 302**中找到）。该对象是*finder*对象的一个列表，通过以模块的名称调用它们的find*module()*方法可以知道如何加载模块。如果模块正好包含在某个包中（由名称中存在的点号表示），那么父包中的_*path*属性将作为find_module() 第二个参数给出（正在导入的模块的名字中，最后一个点号之前的所有内容）。如果某个finder能够找到该模块，它将返回一个*loader*（后面讨论）或者None。

If none of the finders on sys.meta_path are able to find the module then some implicitly defined finders are queried.Implementations of Python vary in what implicit meta path finders are defined.The one they all do define, though, is one that handles sys.path_hooks, sys.path_importer_cache, and sys.path.

The implicit finder searches for the requested module in the “paths” specified in one of two places (“paths” do not have to be file system paths).If the module being imported is supposed to be contained within a package then the second argument passed to find*module(), _path* on the parent package, is used as the source of paths.If the module is not contained in a package then sys.path is used as the source of paths.

Once the source of paths is chosen it is iterated over to find a finder that can handle that path.The dict at sys.path_importer_cache caches finders for paths and is checked for a finder.If the path does not have a finder cached then sys.path_hooks is searched by calling each object in the list with a single argument of the path, returning a finder or raises ImportError.If a finder is returned then it is cached in sys.path_importer_cache and then used for that path entry.If no finder can be found but the path exists then a value of None is stored in sys.path_importer_cache to signify that an implicit, file-based finder that handles modules stored as individual files should be used for that path.If the path does not exist then a finder which always returns None is placed in the cache for the path.

If no finder can find the module then ImportError is raised.Otherwise some finder returned a loader whose load_module() method is called with the name of the module to load (see **PEP 302** for the original definition of loaders).A loader has several responsibilities to perform on a module it loads.First, if the module already exists in sys.modules (a possibility if the loader is called outside of the import machinery) then it is to use that module for initialization and not

a new module.But if the module does not exist in sys.modules then it is to be added to that dict before initialization begins.If an error occurs during loading of the module and it was added to sys.modules it is to be removed from the dict.If an error occurs but the module was already in sys.modules it is left in the dict.

The loader must set several attributes on the module.**name** is to be set to the name of the module.**file** is to be the “path” to the file unless the module is built-in (and thus listed in sys.builtin_module_names) in which case the attribute is not set.If what is being imported is a package then **path** is to be set to a list of paths to be searched when looking for modules and packages contained within the package being imported.**package** is optional but should be set to the name of package that contains the module or package (the empty string is used for module not contained in a package).**loader** is also optional but should be set to the loader object that is loading the module.

If an error occurs during loading then the loader raises ImportError if some other exception is not already being propagated.Otherwise the loader returns the module that was loaded and initialized.

当步骤（1）结束时没有抛出异常，步骤（2）就可以开始。

import语句的第一种形式将局部命名空间中的模块名绑定到模块对象，然后如果有下一个标识符则继续导入。如果模块后面带有as，则as后面的名称将用于模块的局部名称。

from形式不绑定模块的名称：它遍历标识符序列，在步骤（1）中找到的模块中逐一查找它们，然后绑定局部命名空间中的名称到找到的对象。与第一种形式的import类似，可以通过“as localname”提供另外一个名称。如果找不到名称，则引发ImportError。如果用星号('*')替换标识符序列，那么模块中定义的所有公开的名称都将被绑定到import语句所在的局部命名空间。

模块定义的公开的名称通过检查模块命名空间中一个名为**all**的变量决定；如果定义，它必须是一个字符串序列，它们是该模块定义或者导入的名称。**all**中给出的名称都被认为是公开的且要求必须存在。如果**all**没有定义，那么公开的名称集合包括模块命名空间中找到的所有不是以下划线字符 ('')开始的名称。_*all*应该包含全部的公开API。它的意图是避免意外地导出不是API的部分（例如模块内部导入和使用的库模块）。

带有的*from* 形式只可以出现在模块作用域中。如果通配符形式的导入— *import* — 在函数中使用并且函数包含或者是一个带有自由变量的嵌套代码块，编译器将抛出SyntaxError。

在指出你要导入的模块时，你不必指明模块的绝对路径名。当一个模块或者包包含在另外一个包中时，可以在同一个等级的包中使用相对导入而不需要提及包的名字。通过在from之后指定的模块或包中使用前导的点号，你可以指定相对当前包层级向上

7. 复合语句

复合语句包含（多组）其它语句；它们以某种方式影响或者控制其它那些语句的执行。通常，复合语句跨越多行，虽然一条完整的复合语句可以用简洁的形式包含在一行之中。

if、while和for语句实现传统的控制流句法结构。try指出一组语句的异常处理器和/或清理代码。函数和类定义在语法上同样也是复合语句。

复合语句由一个或多个‘子句’组成。一条子句由语句首和‘语句组’组成。一条特定的复合语句的所有子句的语句首都处在相同的缩进水平上。每一个子句的语句首以一个唯一的标识关键字开始并以冒号结束。语句组是由一条子句控制的一组语句。一个语句组可以是语句首冒号之后的同一行上紧跟一个或多个分号分隔的简单语句，也可以是后续行上一个或多个缩进的语句。只有后一种形式的语句组可以包含嵌套的复合语句；下面的语句是非法的，最主要是因为不能明确随后的else子句属于哪一个if子句：

```
if test1: if test2: print x
```

同时要注意在该上下文中分号的优先级比冒号高，所以在下面的例子中，要么执行所有的print语句，要么都不执行：

```
if x < y < z: print x; print y; print z
```

总结：

```
compound_stmt ::=  if_stmt
                   | while_stmt
                   | for_stmt
                   | try_stmt
                   | with_stmt
                   | funcdef
                   | classdef
                   | decorated
suite         ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement     ::=  stmt_list NEWLINE | compound_stmt
stmt_list     ::=  simple_stmt (";" simple_stmt)* [";"]
```

注意语句永远以NEWLINE结束，其后可能跟随一个DEDENT。还要注意可选的续行子句永远以一个不能作为语句开始的关键字开始，因此不会有歧义（‘悬挂的else’问题在Python中通过要求嵌套的if语句必须缩进得到解决）。

为了清晰起见，下面小节中的语法规则的格式会将子句放在单独的一行。

7.1. if 语句

if 语句用于条件执行：

```
if_stmt ::=  "if" expression ":" suite
             ( "elif" expression ":" suite )*
             ["else" ":" suite]
```

它通过对表达式逐个求值直到其中一个为真的方式准确地选择一个语句组（真和假的定义参见布尔操作 一节）；然后执行这个语句组（if语句的其它部分不会被执行或求值）。如果所有表达式都为假，则执行else子句的语句组。

7.2. while 语句

while语句用于重复执行直到某个表达式为真：

```
while_stmt ::=  "while" expression ":" suite
                ["else" ":" suite]
```

它重复测试表达式，如果为真，则执行第一个语句组；如果表达式为假（可能是第一次测试），则执行else子句且终止循环。

第一个语句组中执行的break语句会终止循环而不执行else子句的语句组。在第一个语句组中执行的continue语句会跳过语句组中剩余的语句并返回继续测试表达式。

7.3. for 语句

for语句用于迭代一个序列的元素（例如字符串、元组或者列表）或者其它可迭代的对象：

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

表达式列表值计算一次；它应当产生一个可迭代的对象。expression_list的结果创建一个迭代器。然后以索引的升序为顺序，为迭代器提供的每个元素执行一次语句组。每个元素使用标准的赋值规则被赋值给目标列表，然后执行语句组。当元素取尽时（如果序列为空则立刻取尽），则执行else子句中的语句组并终止循环。

第一个语句组中的break语句会终止循环而不执行else子句的语句组。在第一个语句组中执行的continue语句会跳过语句组的剩余语句并继续下一个元素，如果没有下一个元素则继续执行else子句。

语句组可以给目标列表中的变量赋值；这不影响下一个赋值给它的元素。

当循环结束时目标序列不会被删除，但是如果序列为空，循环不会赋任何值给它。提示：内建的range()函数返回整数的序列，它适用于模拟Pascal 语言中fori:=atobdo效果；例如，range(3) 返回列表[0,1,2]。

注意

当序列被循环修改时，会发生微妙的事情（只有可变类型的序列会发生，例如列表）。有一个内部计数器用于跟踪下一轮循环使用哪一个元素，并且每次迭代中会增加。当计数器达到序列的长度时循环终止。这意味着如果语句组从序列中删除当前（或者前面的）元素，下一个元素将被跳过（因为它获取当前已经被处理过的元素的索引）。同样地，如果语句组在序列中当前元素之前插入一个元素，那么当前元素将在下一次循环被再次处理。这可能导致难以觉察的错误，但可以通过使用整个序列的切片生成临时拷贝避免这个问题，例如，

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4. try 语句

try语句为一组语句指定异常处理器和/或清理代码：

```
try_stmt  ::=  try1_stmt | try2_stmt
try1_stmt ::=  "try" ":" suite
               ("except" [expression [("as" | ",") target]] ":" suite)+
               ["else" ":" suite]
               ["finally" ":" suite]
try2_stmt ::=  "try" ":" suite
               "finally" ":" suite
```

版本2.5 中的新变化：在以前版本的Python 中，try...except...finally 不工作。try...except必须嵌套在try...finally中。

except子句指定一个或多个异常处理器.。当try子句中没有出现异常时，不会执行异常处理器。当try语句组中出现异常时，开始搜索异常处理器。搜索依次检查异常子句直到找到与异常匹配的一个。没有表达式的异常子句，如果出现，必须放在最后；它匹配任何异常。对于一个带有表达式的异常子句，该表达式将被求值，如果结果对象与异常“兼容”，则认为子句与异常匹配。对象与异常兼容，如果它是异常对象的类或基类或者是一个包含与异常兼容的元素的元组。

如果没有except子句匹配到异常，异常处理器的搜索将继续在外层代码和调用栈上进行。[1]

如果计算except子句头部的一个表达式引发了异常， 那么就会中断原异常处理器的搜索， 而在外层代码和调用栈上搜索新的异常处理器（就好像是整个try语句发生了异常一样）。

当找到一个匹配的except子句时，异常就被赋给excep子句中的目标，然后执行excep子句的语句组。所有的异常子句必须具有一个可执行的代码块。当到达该代码块的结尾时，在真个try语句之后执行正常继续。（这意味着如果同一个异常存在两个嵌套的处理器，且异常发生在内部处理器的try子句中，那么外边的处理器不会处理这个异常。）

在执行except子句的语句组之前，异常的详细信息被赋值给sys模块中的三个变量：sys.exc_type接收标识异常的对象；sys.exc_value接收异常的参数；sys.exc_traceback接收一个回溯对象（参见标准类型的层级一节）指示程序中异常发生的点。这些详细信息也可以通过sys.exc_info()函数得到，它返回一个元组(exc_type,exc_value,exc_traceback)。鼓励使用这个函数而不鼓励使用对应的变量，因为在多线程程序中它们的使用是不安全的。从Python 1.5 开始，这些值会在处理异常的函数返回时会恢复它们之前的值（调用之前的值）。

如果控制流从try子句的结尾出来时，则执行可选的else子句。[2]else子句中的异常不会被前面的except子句处理。

如果有finally出现，它指定一个“清除”处理器。首先执行try子句被执行，然后包括任何except和else子句。如果异常发生在任何子句中且没有被处理，那么异常会被临时保存起来。最后执行finally子句。如果有保存的异常，它会在finally子句结束时被重新抛出。如果finally抛出另外一个异常或者执行一个return或break语句，那么保存的异常会被丢弃：

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

在finally子句执行过程中程序访问不到异常信息。

当return、break或continue语句在try...finally语句的try语句组中被执行，finally子句在‘出口’处同样被执行。continue语句出现在finally子句中是非法的。（原因是当前实现的问题 — 该限制在未来可能会去掉）。

函数的返回值取决于执行的最后一条return语句。因为finally子句会永远执行，在finally子句中执行的return语句将永远是最后执行的一条语句：

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

额外的信息可以在异常一节中找到，关于如何使用raise语句产生异常可以在*raise语句*一节中找到。

7.5. with 语句

出现于版本2.5。

with用于和上下文管理器定义的方法一起封装代码块的执行（参见*With语句的上下文管理器*一节）。这允许常见的try...except...finally的用法模式封装起来以方便地重用。

```
with_stmt ::=  "with" with_item ("," with_item)* ":" suite
with_item ::=  expression ["as" target]
```

带有一个“item”的with语句的执行按下面的方式进行：

1. 计算上下文表达式（with_item中给出的表达式）以获得一个上下文管理器。
2. 加载上下文管理器的**exit()**方法留着后面使用。
3. 调用上下文管理器的**enter()**方法。
4. 如果with语句包含一个目标，**enter()**的返回值将赋值给它。

注意

with语句保证如果**enter()**方法没有错误返回，那么**exit()**将永远被调用。因此，如果在给目标列表赋值过程中出现错误，它将被与语句组中发生的错误同样对待。参见下面的第6步。

1. 执行语句组。
2. 调用上下文管理器的**exit()**方法。如果异常导致语句组要退出，那么异常的类型、值和回溯栈被当做参数传递给**exit()**。否则，提供三个None 参数。

如果语句组由于异常退出，且**exit()**方法的返回值为假，异常将被重新引发。如果返回值为真，异常将被取消，并继续执行with语句之后的语句。

如果语句组由于异常以外的其它任何原因退出，**exit()**的返回值将被忽略，执行将在退出发生的正常位置继续。

如果有多个条目，上下文管理器的处理如同嵌套的多个with语句：

```
with A() as a, B() as b:
    suite
```

等同于

```
with A() as a:
    with B() as b:
        suite
```

注意

在Python 2.5中，with只有在with_statement特性被启用的时候才允许使用。在Python 2.6中，它被永远启用。

版本2.7 中的新变化：支持多个上下文表达式。

另请参阅

PEP 0343 - “with”语句Python with语句的说明、背景和实例。

7.6. 函数定义

函数定义定义一个用户自定义的函数对象（参见标准类型的层次一节）：

```
decorated      ::=  decorators (classdef | funcdef)
decorators     ::=  decorator+
decorator      ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
funcdef        ::=  "def" funcname "(" [parameter_list] ")" ":" suite
dotted_name    ::=  identifier ("." identifier)*
parameter_list ::=  (defparameter ",")*
                    (  "*" identifier ["," "**" identifier]
                    | "**" identifier
                    | defparameter [","] )
defparameter   ::=  parameter ["=" expression]
sublist        ::=  parameter ("," parameter)* [","]
parameter      ::=  identifier | "(" sublist ")"
funcname       ::=  identifier
```

函数定义是一个可执行的语句。它的执行将绑定当前局部命名空间中的函数名到一个函数对象（函数可执行代码的封装）。函数对象包含一个对当前全局命名空间的引用，作为函数调用时使用的全局命名空间。

函数定义不会执行函数体；它只有在调用函数的时候才执行。[3]

函数定义可能被一个或多个修饰符表达式封装。修饰符表达式在函数定义时于包含函数定义的定义域中求值。求值的结果必须是一个可调用对象，它以该函数对象为唯一的参数。调用的返回值绑定在函数名而不是函数对象上。多个修饰符是以嵌套的方式作用的。例如，下面的代码：

```
@f1(arg)
@f2
def func(): pass
```

等同于：

```
def func(): pass
func = f1(arg)(f2(func))
```

当一个或多个最上层的参数具有*parameter=expression*的形式时，称该函数具有"默认参数值。"对于具有默认值的参数，对应的参数在调用时可以省略，在这种情况下使用参数的默认值。如果一个参数具有默认值，所有随后的参数也必须具有默认值 — 这个限制在语法中没有表达出来的。

默认的参数值在执行函数定义是求值。这意味着只在函数定义的时候该表达式求一次值，以后每次调用使用相同的"提前计算好的"值。这对于理解默认参数是可变对象时特别重要，例如列表或字典：如果函数修改了该对象（例如，向列表添加一个元素），默认值将受影响被修改。这通常不是想要的。一种变通的方法是使用None作为默认值，然后在函数体中明确地测试它，例如：

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

函数调用的语义在调用一节有更详细的描述。函数调用永远会给参数列表中的所有的参数赋值，无论是位置参数还是关键字参数或默认值。如果出现"*identifier*"的形式，那么它被初始化为一个可以接收任意多余位置参数元组，默认为空元组。如果有"*identifier"的形式，那么它被初识化为一个可以接收任意的多余关键字参数的新的字典，默认值为空字典。

也可以创建匿名函数（没有绑定到某个名称的函数），以在表达式中直接使用。它使用lambda 表达式，在*Lambdas*一节中有详细描述。注意lambda 表达式仅仅是简单的函数定义的简写；"def"语句中定义的函数和lambda 表达式定义的函数一样，可以传递或者赋值给另外一个名称。"def" 形式事实上功能更强大，因为它允许执行多条语句。

程序员的注意事项：函数是第一级的对象。在函数内部执行的“def”形式定义一个可以被返回或传递的局部函数。在嵌套的函数中使用的自由变量可以访问包含该def的函数的局部变量。详细信息参见名称和绑定一节。

7.7. 类定义

类定义定义一个类对象（参见标准类型的层次 一节）：

```
classdef    ::=  "class" classname [inheritance] ":" suite
inheritance ::=  "(" [expression_list] ")"
classname   ::=  identifier
```

类定义是一个可执行语句。它首先计算inheritance序列，如果存在的话。inheritance序列中的每一项都应该是一个类对象或者允许生成子类的类类型。然后使用一个新创建的局部命名空间和初始的全局命名空间，在新的执行帧中执行类的语句组（参见名称和绑定一节）。（通常，语句组只包含函数的定义。）当类的语句组结束执行，它的执行帧被丢弃但是局部命名空间被保存下来。[4]最后使用inheritance序列作为基类创建一个类对象，并保存局部命名空间作为属性字典。类的名称被绑定到初识局部命名空间中类对象。

程序员的注意事项：类定义中定义的变量是类变量；它们被所有的实例共享。要创建实例变量，可以在方法中使用self.name=value设置它们。类变量和实例变量都可以通过“self.name”记号访问，当用相同的方式访问时实例变量将隐藏相同名称的类变量。类变量可以作为实例变量的默认值使用，但是这种用法如果使用可变的值可能导致意想不到的结果。对于新式类，可以使用描述符创建具有不同实现细节的实例变量。

类定义，类似于函数定义，可以被一个或多个描述符 表达式封装。描述符表达式的计算规则和函数相同。结果必须是一个类对象，并绑定于类的名字。

脚注

[1]	异常将扩散到调用栈除非**finally**子句碰巧引发另外一个异常。这个新的异常导致旧的异常丢失。

[2]	目前，控制“从末尾流出”除了下面这些情况：异常或执行**return**、**continue**、**break**语句。

[3]	作为函数体第一条语句出现的字符串字面值被转换成函数的**doc**属性，即函数的文档字符串。

| [4] | 作为类体的第一条语句出现的语句被转换为该命名空间的**doc**属性，即类的文档字符串。 | |-----|-----|

8. 顶层的组件

Python 解释器可以从多个源获取输入：从以标准输入或者程序参数传递给它的脚本，交互式输入，模块源文件等。本章给出这些情况下使用的语法。

8.1. 完整的Python 程序

虽然语言的规范不需要规定语言的解释器如何被调用，但是对完整的Python 程序有一个概念是很有用的。一个完整的Python 程序在一个最小初始化的环境中执行：可以访问所有的内建和标准模块，但是，除了sys（各种系统服务），**builtin**（内建函数，异常和 None) 以及**main**，都没有初始化。后者用来给完整的程序的执行提供局部和全局命名空间。

完整的Python 程序的语法用于文件输入，在下面的小节中讲述。

解释器可能也会在交互模式下被调用；在这种情况下，它不会读取并执行一个完整的程序，但是它会一次读取执行一条语句（可以是复合语句）。它的初识环境和完整的程序是完全一样的；每一条语句在**main**命名空间中执行。

在Unix下，一个完整的程序可用三种形式传递给解释器：带有-c*string* 的命令行选项，以文件传递的第一个命令行参数，或者以标准输入。如果文件或者标准输入是一个tty 设备，解释器将进入交互模式；否则，它执行文件作为一个完整的程序。

8.2. 文件输入

所有从非交互式文件读取的输入都具有相同的形式：

```
file_input ::=  (NEWLINE | statement)*
```

该语法用在以下的情形：

- 当解析一个完整的Python 程序（从一个文件或者一个字符串）；
- 当解析一个模块；
- 当解析一个传递给exec语句的字符串；

8.3. 交互式输入

交互模式下的输入使用下面的语法解析：

```
interactive_input ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

注意（顶层）组件的语句在交互模式下后面必须跟随一个空格；它可以帮助解析器检测到输入的结束。

8.4. 表达式输入

有两种形式的表达式输入。两种形式都忽略前导的空格。eval()的字符串参数必须具有以下的形式：

```
eval_input ::=  expression_list NEWLINE*
```

由input()读取的输入行必须具有以下的形式：

```
input_input ::=  expression_list NEWLINE
```

注意：为了读取‘原始’的不用解释的输入行， 你可以使用内建的函数raw_input() 或者文件对象的readline()。

9. 完整的语法规范

这是完整的Python语法，它由解析器读入用于解析Python源文件：

```
# Grammar for Python

# Note:  Changing the grammar specified in this file will most likely
#        require corresponding changes in the parser module
#        (../Modules/parsermodule.c).  If you can't make the changes to
#        that module yourself, please co-ordinate the required changes
#        with someone who can; ask around on python-dev for help.  Fred
#        Drake <fdrake@acm.org> will probably be listening there.

# NOTE WELL: You should also follow all the steps listed in PEP 306,
# "How to Change Python's Grammar"

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() and input() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
```

```
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test [',' test [',' test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classde
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test

test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
```

```
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [','])) )

classdef: 'class' NAME ['(' [testlist] ')'] ':' suite

arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',' test)*

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [testlist]
```

Python 3 教程

Python 是一门简单易学、功能强大的编程语言。它具有高效的高级数据结构和简单而有效的面向对象编程方法。Python 优雅的语法和动态类型、 以及其解释性的性质，使它在许多领域和大多数平台成为脚本编写和快速应用程序开发的理想语言。

从 Python 网站 http://www.python.org/可以免费获得所有主要平台的源代码或二进制形式的 Python 解释器和广泛的标准库，并且可以自由地分发。网站还包含许多免费的第三方 Python 模块、 程序、工具以及附加文档的发布包和链接。

Python 解释器可以用 C 或 C+ + （或可从 C 中调用的其他语言） 中实现的新的函数和数据类型轻松扩展。Python 也适合作为可定制应用程序的一种扩展语言。

本教程非正式向读者介绍 Python 语言及其体系的基本概念和功能。 随手使用 Python 解释器来亲自动手是很有帮助的，但所有示例都是自包含的，所以本教程也可以离线阅读。

有关标准对象和模块的说明，请参阅*Python 标准库*。*Python语言参考* 给出了Python语言的更正式的定义。要编写 C 或 C+ + 的扩展，请阅读*扩展和嵌入Python解释器* 与*Python/C API参考手册*。也有几本书深度地介绍了Python 。

本教程不会尝试全面地涵盖每一个单独特性，甚至即使它是常用的特性。相反，它介绍了许多 Python 的值得注意的特性，从而能让你很好的把握这门语言的特性。经过学习，你将能够阅读和编写Python 的模块和程序，并可以更好的学会 *Python 标准库* 中描述的各种 Python 库模块。

词汇表 也值得浏览一下。

1. 引言

如果你要用计算机做很多工作，最终你发现是有一些您希望自动执行的任务。例如，你可能希望对大量的文本的文件执行搜索和替换，或以复杂的方式重命名并重新排列一堆照片文件。也许你想写一个小的自定义数据库，或一个专门的 GUI 应用程序或一个简单的游戏。

如果你是一个专业的软件开发人员，您可能必须使用几个 C / C + + /Java 库，但发现通常的编写/编译/测试/重新编译周期太慢。也许你要写这样的库中的测试套件，然后发现编写测试代码是很乏味的工作。或也许您编写了一个程序，它可以使用一种扩展语言，但你不想为您的应用程序来设计与实现一个完整的新语言。

Python 正是这样为你准备的语言。

你可以为其中一些任务写一个 Unix shell 脚本或 Windows 批处理文件，但是 shell 脚本最适合处理文件移动和文本编辑，而不适用于 GUI 应用程序和游戏。你可以写一个 C / C + +/Java 程序，但是甚至程序的第一个初稿都可能花费大量的开发时间。Python 更简单易用，可用于 Windows、 Mac OS X 和 Unix 操作系统，并将帮助您更快地完成工作。

Python 使用很简单，但它是一个真正的编程语言，比 shell 脚本或批处理文件对于大型的程序提供更多的结构和支持。另一方面，Python 还提供了比 C 更多的错误检查，并且，作为一种高级语言，它有内置的高级数据类型，比如灵活的数组和字典。因为其更加一般的数据类型， Python 比 Awk 甚至 Perl 适用于很多更大的问题领域，而且很多事情在 Python 中至少和那些语言一样容易。

Python 允许您将您的程序拆分成可以在其他 Python 程序中重复使用的模块。它拥有大量的标准模块，你可以将其用作你的程序的基础 — 或者作为学习 Python 编程的示例。这些模块提供诸如文件 I/O、 系统调用、 套接字和甚至用户图形界面接口，例如Tk。

Python 是一门解释性的语言，因为没有编译和链接，它可以节省你程序开发过程中的大量时间。Python 解释器可以交互地使用，这使得试验Python语言的特性、编写用后即扔的程序或在自底向上的程序开发中测试功能非常容易。它也是一个方便的桌面计算器。

Python 使程序编写起来能够紧凑和可读。编写的 Python 程序通常比等价的 C、 C + + 或 Java 程序短很多，原因有几个：

- 高级数据类型允许您在单个语句中来表达复杂的操作；
- 语句分组是通过缩进，而不是开始和结束的括号 ；
- 任何变量或参数的声明不是必要的。

Python 是可扩展的: 如果你知道如何用 C 编程，那么将很容易添加一个新的内置函数或模块到解释器中，要么为了以最快的速度执行关键的操作，要么为了将 Python 程序与只有二进制形式的库（如特定供应商提供的图形库）链接起来。一旦你真的着迷，你可以把 Python 解释器链接到 C 编写的应用程序中，并把它当作那个程序的扩展或命令行语言。

顺便说一句，Python 语言的名字来自于BBC 的 “Monty Python’s Flying Circus” 节目，与爬行动物无关。在文档中引用Monty Python 短剧不仅可以，并且鼓励！

既然现在你们都为 Python 感到兴奋，你们一定会想更加详细地研究它。学习一门语言最好的方法就是使用它，本教程推荐你边读边使用 Python 解释器练习。

在下一章中，我们将解释 Python 解释器的用法。这是很简单的一件事情，但它有助于试验后面的例子

本教程的其余部分通过实例介绍了 Python 语言和体系的各种特性，以简单的表达式、 语句和数据类型开始，然后是 函数和模块，最后讲述高级概念，如异常和用户自定义的类。

2. Python 解释器

2.1 调用解释器

Python 解释器在可用的机器上通常安装成/usr/local/bin/python3.4；将/usr/local/bin放在您的Unix shell 搜索路径，使得可以通过在shell 中键入命令

```
python3.4
```

来启动它。[1]由于解释器放置的目录是一个安装选项，其他地方也是可能的；请与您的Python 专家或系统管理员联系。（例如， /usr/local/python是一个常见的替代位置。

在Windows机器上，Python的安装通常是放置在C:\Python3，当然你可以在运行安装程序时进行更改。你可以在一个DOS窗口的命令提示符下键入以下命令来把这个目录添加到路径中：

```
set path=%path%;C:\python34
```

主提示符下键入文件结束字符 （ Unix 上是Control-D 、 Windows 上是Control-Z） 会导致该解释器以零退出状态退出。如果无法正常工作，您可以通过键入以下命令退出解释器：quit()。

编辑器器的行编辑功能包括交互式编辑，历史记录和代码补全，其中代码补全功能需要系统支持readline库。也许最快的检查，看看是否支持命令行编辑对你的第一个 Python 提示Ctrl-P。如果它发出蜂鸣声，则有命令行编辑 ；请参阅附录交互式输入编辑和历史替代 的有关快捷键的介绍。如果什么都没发生，或者显示^ P，命令行编辑不可用 ；你就只能够使用退格键删除当前行中的字符。

解释器有些像 Unix shell：当使用tty设备作为标准输入调用时，它交互地读取并执行命令 ；当用文件名参数或文件作为标准输入调用，它将读取并执行该文件中的脚本。

第二种启动解释器的方式是python-ccommand[arg]...，这种方式是在*command*中执行语句，类似于 shell 的*-c*选项中。因为 Python 语句经常包含空格或其他shell 特殊字符，通常建议把全部命令放在单引号里。

有些 Python 模块同时也是可执行的脚本。这些模块可以使用python-mmodule[arg]......直接调用，这和在命令行输入完整的路径名执行模块 的源文件是一样的。

当使用一个脚本文件时，它有时是很有用能够运行该脚本，之后进入交互模式。这可以通过在脚本前面加上*-i* 选项实现。

2.1.1. 参数传递

调用解释器时，脚本名称和其他参数被转换成一个字符串列表并赋值给sys模块中的argv变量。你可以通过importsys访问此列表。列表的长度是至少是 1 ；如果没有给出脚本和参数，sys.argv[0]是一个空字符串。当使用*-c*命令时， sys.argv[0]设置为'-c'。当使用*-m*模块参数时， sys.argv[0]被设定为指定模块的全名。*-c*选项或*-m*选项后面的选项不会被Python 解释器处理，但是会被保存在sys.argv中，供命令或模块使用。

2.1.2. 交互模式

当从 tty 读取命令时，我们说解释器在交互模式 下。这种模式下，解释器以主提示符 提示输入命令，主提示符通常标识为三个大于号(>>>） ；如果有续行，解释器以从提示符 提示输入，默认为三个点 (...)。在第一个提示符之前，解释器会打印出一条欢迎信息声明它的版本号和授权公告：

```
$ python3.4
Python 3.4 (default, Mar 16 2014, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行结构时需要续行。作为一个例子，看看这个if语句：

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

2.2. 解释器及其环境

2.2.1. 错误处理

错误发生时，解释器会打印错误信息和堆栈跟踪信息。在交互模式下，它会返回到主提示符；当输入来自一个文件，它会打印堆栈跟踪信息，然后以非零退出状态退出。（由try语句中except的子句处理的异常不是在这方面的错误）。有些错误是致命的并导致非零状态退出；这通常由于内部不一致和某些情况下的内存不足导致。所有错误消息都写入标准错误流；执行命令的普通输出写入标准输出。

在主提示符或从提示符后输入中断符(通常为Control-C或DEL)可以取消输入，并返回到主提示符。[2]命令执行过程中输入中断符将引发KeyboardInterrupt异常，它可以被try语句截获。

2.2.2. 可执行的 Python 脚本

在类 BSD 的 Unix 系统上，可以将Python 脚本变成可直接执行的，就像 shell 脚本一样。通过放置一行

```
#! /usr/bin/env python3.4
```

（假定解释器在用户的PATH 中） 在脚本的开始并且给文件可执行的模式。#!必须是文件的前两个字符。在一些平台上，这第一行必须以一个 Unix 风格的行结束符 ('\n') ， 而不是 Windows 的行结束符 （'\r\n') 结尾。注意，字符 '#'，是Python 注释的起始符号。

可以通过 **chmod** 命令给予脚本可执行的模式或权限：

```
$ chmod +x myscript.py
```

在 Windows 系统上，没有"可执行模式"的概念。Python 安装程序会自动将 .py 文件与 python.exe 关联，双击 Python 文件将以脚本的方式运行它。扩展名也可以是 .pyw，在这种情况下，通常出现的控制台窗口不会在显示了。

2.2.3. 源程序的编码

Python源文件默认以UTF-8编码。在这种编码下，世界上大多数语言的字符可以在字符串，标识符和注释中同时使用 — 尽管标准库中的标识符只使用ASCII字符，它是可移植代码应该遵循的一个惯例。为了能够正确显示所有的这些字符，你的编辑器必须能够识别文件是UTF-8编码，且必须使用支持文件中所有字符的字体。

也可以给源文件指定一个不同的编码。方法是在 #! 行的后面再增加一行特殊的注释来定义源文件的编码：

```
# -*- coding: encoding -*-
```

通过此声明，源文件中的所有字符将被视为由 *encoding* 而不是UTF-8编码。在 Python 库参考手册的 codecs 小节中，可以找到所有可能的编码方式列表。

例如，如果你选择的编辑器不支持UTF-8编码的文件，而只能用其它编码比如Windows-1252，你可以这样写：

```
# -*- coding: cp-1252 -*-

currency = u"€"
print ord(currency)
```

并且源文件中的所有字符仍然使用Windows-1252字符集。这个特殊的编码注释必须位于文件的第一或者第二 行。

2.2.4. 交互式启动文件

当您以交互方式使用 Python 时，让解释器在每次启动时执行一些标准命令会变得非常方便。你可以通过设置环境变量 PYTHONSTARTUP 为包含你的启动命令的文件的名字。这类似于 Unix shell 的.profile功能。

这个文件只会在交互式会话时读取，当 Python 从脚本中读取命令时不会读取，当/dev/tty 在命令中明确指明时也不会读取（尽管这种方式很像是交互方式）。它和交互式命令在相同的命名空间中执行，所以在交互式会话中，由它定义或引用的一切可以在解释器中不受限制地使用。您还可以在此文件中更改sys.ps1和sys.ps2 的提示符。

如果您想要从当前目录读取额外的启动文件，你可以在全局启动文件中使用这样的代码 ifos.path.isfile('.pythonrc.py'):exec(open('.pythonrc.py').read())。如果你想要在脚本中使用启动文件，必须要在脚本中明确地写出：

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    exec(open(filename).read())
```

2.2.5. 自定义模块

Python 提供了两个钩子（方法）来本地化： sitecustomize和usercustomize。若要查看它是如何工作，你首先需要找到你的 site-packages 的目录。启动 Python 并运行下面的代码：

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.4/site-packages'
```

现在你可以在此目录下创建名为usercustomize.py的文件，并把任何你想要的东西放在里面。它会影响每个 Python 调用，除非启动时用-s选项来禁用自动导入。

sitecustomize的工作方式相同，但通常由计算机的管理员在全局 site-packages 目录中创建，并在usercustomize之前导入。更多详细信息请参阅 site 模块的文档。

脚注

[1]	**On Unix, the Python 3.x interpreter is by default not installed with the executable named python, so that it does not conflict with a simultaneously installed Python 2.x executable.**

| [2] | A problem with the GNU Readline package may prevent this. | |-----|-----|

3. Python简介

以下的示例中，输入和输出通过有没有以提示符（>>>和...）来区分：如果要重复该示例，你必须在提示符出现后，输入提示符后面的所有内容；没有以提示符开头的行是解释器的输出。注意示例中出现从提示符意味着你一定要在最后加上一个空行；这用于结束一个多行命令。

本手册中的很多示例，甚至在交互方式下输入的示例，都带有注释。Python 中的注释以“井号”， #，开始，直至实际的行尾。注释可以从行首开始，也可以跟在空白或代码之后，但不能包含在字符串字面量中。字符串字面量中的#字符仅仅表示#。因为注释只是为了解释代码并且不会被Python解释器解释，所以敲入示例的时候可以忽略它们。

例如：

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. 将 Python 当做计算器

让我们尝试一些简单的 Python 命令。启动解释器然后等待主提示符 >>>。（应该不需要很久。）

3.1.1. 数字

解释器可作为一个简单的计算器：你可以向它输入一个表达式，它将返回其结果。表达式语法很直白: 运算符+、 -、 *和/的用法就和其它大部分语言一样（例如 Pascal 或 C）；括号(())可以用来分组。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

整数(例如 2, 4, 20)的类型是int，带有小数部分数字(e.g. 5.0, 1.6)的类型是float。在本教程的后面我们会看到更多关于数字类型的内容。

除法(/)永远返回一个浮点数。如要使用*floor* 除法并且得到整数结果（丢掉任何小数部分），你可以使用//运算符；要计算余数你可以使用%：

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

通过Python，还可以使用**运算符符计算幂乘方[1]：

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

等号 (=) 用于给变量赋值。赋值之后，在下一个提示符之前不会有任何结果显示：

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

如果变量没有“定义”（赋值），使用的时候将会报错：

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

浮点数完全支持；整数和浮点数的混合计算中，整数会被转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

在交互模式下，最近一次表达式的值被赋给变量_。这意味着把 Python 当做桌面计算器使用的时候，可以方便的进行连续计算，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

用户应该将这个变量视为只读的。不要试图去给它赋值 —— 你将会创建出一个独立的同名局部变量，并且屏蔽了内置变量的魔术效果。

除了int 和 float，Python还支持其它数字类型，例如小数 and 分数。Python还内建支持复数，使用后缀j 或 J 表示虚数部分（例如3+5j）。

3.1.2. 字符串

除了数值，Python 还可以操作字符串，可以用几种方法来表示。它们可以用单引号('...')或双引号("...")括起来，效果是一样的[2]。\ 可以用来转义引号。

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

在交互式解释器中，输出的字符串会用引号引起来，特殊字符会用反斜杠转义。虽然可能和输入看上去不太一样，但是两个字符串是相等的。如果字符串中只有单引号而没有双引号，就用双引号引用，否则用单引号引用。print()函数生成可读性更好的输出, 它会省去引号并且打印出转义后的特殊字符：

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

如果你前面带有\的字符被当作特殊字符，你可以使用原始字符串，方法是在第一个引号前面加上一个r:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

字符串可以跨多行。一种方法是使用三引号："""..."""或者'''...'''。行尾换行符会被自动包含到字符串中，但是可以在行尾加上 \ 来避免这个行为。下面的示例：

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

将生成以下输出（注意，没有开始的第一行）：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

字符串可以用 +操作符联接，也可以用* 操作符重复多次：

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

相邻的两个或多个字符串字面量（用引号引起来的）会自动连接。

```
>>> 'Py' 'thon'
'Python'
```

然而这种方式只对两个字面量有效，变量或者表达式是不行的。

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

如果你想连接多个变量或者连接一个变量和一个常量，使用+：

```
>>> prefix + 'thon'
'Python'
```

这个功能在你想输入很长的字符串的时候特别有用：

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

字符串可以索引，第一个字符的索引值为0。Python 没有单独的字符类型；字符就是长度为 1 的字符串。

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

索引也可以是负值，此时从右侧开始计数：

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

注意，因为 -0 和 0 是一样的，负的索引从 -1 开始。

除了索引，还支持切片。索引用于获得单个字符，切片让你获得子字符串：

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

注意，包含起始的字符，不包含末尾的字符。这使得s[:i]+s[i:] 永远等于 s：

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

切片的索引有非常有用的默认值；省略的第一个索引默认为零，省略的第二个索引默认为切片的字符串的大小。

```
>>> word[:2]  # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]  # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

有个方法可以记住切片的工作方式，把索引当做字符之间的点，第一个字符的左边是0。含有 *n* 个字符的字符串的最后一个字符的右边是索引 *n*，例如：

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行给出了字符串中 0..5 各索引的位置；第二行给出了相应的负索引。从 *i* 到 *j* 的切片由 *i* 和 *j* 之间的所有字符组成。

对于非负索引，如果上下都在边界内，切片长度就是两个索引之差。例如，word[1:3] 的长度是 2。

试图使用太大的索引会导致错误：

```
>>> word[42]  # the word only has 7 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

但是，当用于切片时，超出范围的切片索引会优雅地处理：

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python 字符串不可以改变—它们是不可变的。因此，赋值给字符串索引的位置会导致错误：

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

如果你需要一个不同的字符串，你应该创建一个新的：

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

内置函数len()返回字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

请参阅

*Text Sequence Type — str*Strings are examples of *sequence types*, and support the common operations supported by such types.*String Methods*Strings support a large number of methods for basic transformations and searching.*String Formatting*Information about string formatting with str.format() is described here.*printf-style String Formatting*The old formatting operations invoked when strings and Unicode strings are the left operand of the % operator are described in more detail here.

3.1.3. 列表

Python 有几个 复合 数据类型，用来组合其他的值。最有用的是 列表，可以写成中括号中的一列用逗号分隔的值。列表可以包含不同类型的元素，但是通常所有的元素都具有相同的类型。

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

和字符串（以及其它所有内建的 序列类型）一样，列表可以索引和切片：

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

所有的切片操作都会返回一个包含请求的元素的新列表。这意味着下面的切片操作返回列表一个新的（浅）拷贝副本。

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

列表也支持连接这样的操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

与不可变的字符串不同，列表是可变的 类型，例如可以改变它们的内容：

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

你还可以使用append()方法（后面我们会看到更多关于方法的内容）在列表的末尾添加新的元素：

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

给切片赋值也是可以的，此操作甚至可以改变列表的大小或者清空它: :

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

内置函数len()也适用于列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

列表可以嵌套 （创建包含其他列表的列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. 编程第一步

当然，我们可以将 Python 用于比 2 加 2 更复杂的任务。例如，我们可以写一个生成斐波那契初始子序列的程序，如下所示：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

本示例介绍了几种新功能。

- 第一行包括了一个多重赋值：变量a和b同时获得新的值 0 和 1。最后一行又这样使用了一次，说明等号右边的表达式在赋值之前首先被完全解析。右侧表达式是从左到右计算的。
- 只要条件（这里是 b<10）为 true，[while](#)循环反复执行。在="" Python="" 中，和="" C="" 一样，任何非零整数值都为="" true；零为="" false。循环条件也可以是一个字符串或者列表，实际上可以是任何序列；长度不为零的序列为="" true，空序列为="" false。示例中使用的测试是一个简单的比较。标准比较运算符与="" 的写法一样：<（小于），="">（大于），==(等于)，<=（小于或等于），>=（大于或等于）和!=（不等于）。
- 循环体 是缩进 的：缩进是 Python 分组语句的方式。交互式输入时，你必须为每个缩进的行输入一个 tab 或（多个）空格。实践中你会用文本编辑器来编写复杂的 Python 程序；所有说得过去的文本编辑器都有自动缩进的功能。交互式输入复合语句时，最后必须在跟随一个空行来表示结束（因为解析器无法猜测你什么时候已经输入最后一行）。注意基本块内的每一行必须按相同的量缩进。
- print()函数输出传给它的参数的值。与仅仅输出你想输出的表达式不同（就像我们在前面计算器的例子中所做的），它可以输出多个参数、浮点数和字符串。打印出来的字符串不包含引号，项目之间会插入一个空格，所以你可以设置漂亮的格式，像这样：

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

关键字参数 end 可以避免在输出后面的空行，或者可以指定输出后面带有一个不同的字符串：

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Footnotes

[1]	**Since has higher precedence than -, -32 will be interpreted as -(32) and thus result in -9. To avoid this and get 9, you can use (-3)2.**

| [2] | Unlike other languages, special characters such as \n have the same meaning with both single ('...') and double ("...") quotes. The only difference between the two is that within single quotes you don’t need to escape " (but you have to escape \') and vice versa. | |-----|-----|

4. 控制流

除了前面介绍的 while语句，Python 也有其它语言常见的流程控制语句，但是稍有不同。

4.1. if 语句

也许最知名的语句类型是if语句。例如：

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

可以有零个或多个 elif 部分，else 部分是可选的。关键字 'elif' 是 'else if' 的简写，可以有效避免过深的缩进。if...elif ...elif ... 序列用于替代其它语言中 switch 或case 语句。

4.2. for 语句

Python 中的for语句和你可能熟悉的 C 或 Pascal 中的有点不同。和常见的依据一个等差数列迭代（如 Pascal），或让用户能够自定义迭代步骤和停止条件（如 C）不一样，Python 的 for 语句按照元素出现的顺序迭代任何序列（列表或字符串）。例如（没有双关意）：

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

如果要在循环内修改正在迭代的序列（例如，复制所选的项目），建议首先制作副本。迭代序列不会隐式地创建副本。 使用切片就可以很容易地做到：

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3. range() 函数

如果你确实需要遍历一个数字序列，内置函数range()很方便。它会生成等差序列：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

给定的终点永远不会在生成的序列中；若要依据索引迭代序列，你可以结合使用range()和len()，如下所示：也可以让 range 函数从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为'步长'）：

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

若要依据索引迭代序列，你可以结合使用range () 和len()，如下所示：

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

然而，在这种情况下，大部分时候使用enumerate()函数会更加方便，请参见*Looping Techniques*。

如果你只打印range，会出现奇怪的结果：

```
>>> print(range(10))
range(0, 10)
```

range()返回的对象的行为在很多方面很像一个列表，但实际上它并不是列表。当你迭代它的时候它会依次返回期望序列的元素，但是它不会真正产生一个列表，因此可以节省空间。

我们把这样的对象称为可迭代的，也就是说它们适合期望连续获得元素的函数和构造器,直到穷尽。我们已经看到for语句是这样的一个迭代器。list()函数是另外一个；它从可迭代对象创建列表。

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

后面我们会看到更多返回可迭代对象和以可迭代对象作为参数的函数。

4.4. break 和 continue 语句，以及循环中 else 字句

Break 语句和 C 中的类似，用于跳出最近的for 或 while 循环。

循环语句可以有一个 else 子句；当 (for) 循环迭代完整个列表或（while）循环条件变为 false，而非由break 语句终止时，它会执行。下面循环搜索质数的代码例示了这一点：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

（是的，这是正确的代码。看仔细：else子句属于for 循环，不属于 if 语句。）

当使用一个循环， else子句已更像的else子句的try语句而不是，if语句： try语句的else子句时未发生任何异常，和一个循环else子句运行不会中断发生时运行。

更多关于try语句和异常的内容，请参见处理异常。 continue 语句，也是从 C 语言借来的，表示继续下一次迭代：

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

5. 数据结构

本章详细讲述你已经学过的一些知识，并增加一些新内容。

5.1. 深入列表

列表数据类型还有更多的方法。这里是列表对象的所有方法：

list.append(*x*) 添加一个元素到列表的末尾。相当于a[len(a):]=[x]。

list.extend(*L*) 将给定列表中的所有元素附加到另一个列表的末尾。相当于a[len(a):]=L。

list.insert(*i*, *x*) 在给定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，所以 a.insert(0,x) 在列表的最前面插入，a.insert(len(a),x) 相当于 a.append(x)。

list.remove(*x*) 删除列表中第一个值为 *x* 的元素。如果没有这样的元素将会报错。

list.pop([*i*]) 删除列表中给定位置的元素并返回它。如果未指定索引，a.pop() 删除并返回列表中的最后一个元素。（*i* 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示法)。

list.clear() 删除列表中所有的元素。相当于dela[:]。

list.index(*x*) 返回列表中第一个值为 *x* 的元素的索引。如果没有这样的元素将会报错。

list.count(*x*) 返回列表中 *x* 出现的次数。

list.sort(*cmp=None*, *key=None*, *reverse=False*) 原地排序列表中的元素。

list.reverse() 原地反转列表中的元素。

list.copy() 返回列表的一个浅拷贝。等同于a[:]。

使用了列表大多数方法的例子：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

你可能已经注意到像insert, remove 或者 sort之类的方法只修改列表而没有返回值打印出来 -- 它们其实返回了默认值None。[1]这是 Python 中所有可变数据结构的设计原则。

5.1.1. 将列表作为堆栈使用

列表方法使得将列表当作堆栈非常容易，最先进入的元素最后一个取出（后进先出）。使用 append() 将元素添加到堆栈的顶部。使用不带索引的 pop() 从堆栈的顶部取出元素。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. 将列表当作队列使用

也可以将列表当作队列使用，此时最先进入的元素第一个取出（先进先出）；但是列表用作此目的效率不高。在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素必须向后移一位)。

如果要实现一个队列，可以使用 collections.deque，它设计的目的就是在两端都能够快速添加和弹出元素。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. 列表解析

列表解析提供了一个生成列表的简洁方法。应用程序通常会从一个序列的每个元素的操作结果生成新的列表，或者生成满足特定条件的元素的子序列。

例如，假设我们想要创建一个列表 squares：

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

我们可以用下面的方式得到同样的结果：

```
squares = [x**2 for x in range(10)]
```

这也相当于squares=list(map(lambdax:x**2,range(10)))，但是更简洁和易读。

列表解析由括号括起来，括号里面包含一个表达式，表达式后面跟着一个for语句，后面还可以接零个或更多的 for 或 if 语句。结果是一个新的列表，由表达式依据其后面的 for 和 if 字句上下文计算而来的结果构成。例如，下面的 listcomp 组合两个列表中不相等的元素：

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

它等效于：

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意在两个代码段中for 和 if 语句的顺序是相同的。

如果表达式是一个元组（例如前面示例中的 (x, y)），它必须带圆括号。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表解析可以包含复杂的表达式和嵌套的函数：

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. 嵌套的列表解析

列表解析中的第一个表达式可以是任何表达式，包括列表解析。

考虑下面由三个长度为 4 的列表组成的 3x4 矩阵：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

下面的列表解析将转置行和列：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

正如我们在上一节中看到的，嵌套的 listcomp 在跟随它之后的for 字句中计算，所以此例等同于：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以此下去，还等同于：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在实际中，与复杂的控制流比起来，你应该更喜欢内置的函数。针对这种场景，使用 zip() 函数会更好：

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

关于本行中使用的星号的说明，请参阅参数列表的分拆。

5.2. del语句

有个方法可以从列表中按索引而不是值来删除一个元素： del 语句。这不同于有返回值的 pop() 方法。del 语句还可以用于从列表中删除切片或清除整个列表（我们是将空列表赋值给切片）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del 也可以用于删除整个变量：

```
>>> del a
```

此后再引用名称 a 将会报错（直到有另一个值被赋给它）。稍后我们将看到 del 的其它用途。

5.3. 元组和序列

我们已经看到列表和字符串具有很多共同的属性，如索引和切片操作。它们是 序列 数据类型的两个例子(参见 *Sequence Types — list, tuple, range*)。因为 Python 是一个正在不断进化的语言，其他的序列类型也可能被添加进来。还有另一种标准序列数据类型：元组。

元组由逗号分割的若干值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构；在输入时括号可有可无，不过括号经常都是必须的（如果元组是一个更大的表达式的一部分）。不能给元组中单独的一个元素赋值，不过可以创建包含可变对象（例如列表）的元组。

虽然元组看起来类似于列表，它们经常用于不同的场景和不同的目的。元组是不可变的，通常包含不同种类的元素并通过分拆（参阅本节后面的内容) 或索引访问（如果是namedtuples，甚至可以通过属性）。列表是 可变 的，它们的元素通常是相同类型的并通过迭代访问。

一个特殊的情况是构造包含 0 个或 1 个元素的元组：为了实现这种情况，语法上有一些奇怪。空元组由一对空括号号创建；只有一个元素的元组由值后面跟随一个逗号创建 (在括号中放入单独一个值还不够）。丑陋，但是有效。例如：

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句t=12345,54321,'hello!' 是一个元组封装的例子： 值12345,54321 和 'hello!' 被一起放入一个元组。其逆操作也是可以的：

```
>>> x, y, z = t
```

这被称为 序列分拆 再恰当不过了，且可以用于右边的任何序列。序列分拆要求等号左侧的变量和序列中的元素的数目相同。注意多重赋值只是同时进行元组封装和序列分拆。

5.4. 集合

Python 还包含了一个数据类型 集合。集合中的元素不会重复且没有顺序。集合的基本用途有成员测试和消除重复的条目。集合对象还支持并集、 交集、 差和对称差等数学运算。

花大括号或 set() 函数可以用于创建集合。注意： 若要创建一个空集必须使用set()，而不能用 {}；后者将创建一个空的字典，一个我们在下一节中讨论数据结构。

这里是一个简短的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

和 列表解析 类似，Python 也支持集合解析：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. 字典

Python 中内置的另一种有用的数据类型是字典（见映射的类型 —— 字典)。有时候你会发现字典在其它语言中被称为 “associative memories” 或者 “associative arrays”。与序列不同，序列由数字做索引，字典由 键 做索引，键可以是任意不可变类型；字符串和数字永远可以拿来做键。如果元组只包含字符串、 数字或元组，它们可以用作键；如果元组直接或间接地包含任何可变对象，不能用作键。不能用列表作为键，因为列表可以用索引、切片或者 append() 和extend() 方法修改。

理解字典的最佳方式是把它看做无序的 键:值 对集合，要求是键必须是唯一的（在同一个字典内）。一对大括号将创建一个空的字典：{}。大括号中由逗号分隔的 键:值 对将成为字典的初始值；打印字典时也是按照这种方式输出。

字典的主要操作是依据键来存取值。也可以通过 del 删除 键: 值 对。如果用一个已经存在的键存储值，以前为该关键字分配的值就会被遗忘。用一个不存在的键中读取值会导致错误。

list(d.keys())返回字典中所有键组成的列表，列表的顺序是随机的（如果你想它是有序的，只需使用sorted(d.keys())代替）。[2]要检查某个键是否在字典中，可以使用 in 关键字。

下面是一个使用字典的小示例：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

dict() 构造函数直接从键-值对序列创建字典：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典解析可以用于从任意键和值表达式创建字典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果键都是简单的字符串，有时通过关键字参数指定 键-值 对更为方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6. 遍历的技巧

循环迭代字典的时候，键和对应的值通过使用items()方法可以同时得到。

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

序列中遍历时，使用 enumerate() 函数可以同时得到索引和对应的值。

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时遍历两个或更多的序列，使用 zip() 函数可以成对读取元素。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先正向生成这个序列，然后调用 reversed() 函数。

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

循环一个序列按排序顺序，请使用sorted()函数，返回一个新的排序的列表，同时保留源不变。

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

若要在循环内部修改正在遍历的序列（例如复制某些元素），建议您首先制作副本。在序列上循环不会隐式地创建副本。切片表示法使这尤其方便：

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.7. 深入条件控制

while 和 if 语句中使用的条件可以包含任意的操作，而不仅仅是比较。

比较操作符 in 和 notin 检查一个值是否在一个序列中出现（不出现）。is 和 isnot 比较两个对象是否为同一对象；这只和列表这样的可变对象有关。所有比较运算符都具有相同的优先级，低于所有数值运算符。

可以级联比较。例如， a<b= =c 测试 a 是否小于 b 并且 b 等于 c。

可将布尔运算符 and 和 or 用于比较，比较的结果（或任何其他的布尔表达式）可以用 not 取反。这些操作符的优先级又低于比较操作符；它们之间，not 优先级最高，or 优先级最低，所以 Aand notBorC 等效于 (Aand(notB))orC。与往常一样，可以使用括号来表示所需的组合。

布尔运算符and 和 or 是所谓的 短路 运算符：依参数从左向右求值，结果一旦确定就停止。例如，如果A 和 C 都为真，但B是假， AandBandC 将不计算表达式 C。用作一个普通值而非逻辑值时，短路操作符的返回值通常是最后一个计算的。

可以把比较或其它逻辑表达式的返回值赋给一个变量。例如，

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

注意 Python 与 C 不同，在表达式内部不能赋值。C 程序员可能会抱怨这一点，但它避免了一类 C 程序中常见的问题： 在表达式中输入 = 而真正的意图是== 。

5.8. 序列和其它类型的比较

序列对象可以与具有相同序列类型的其他对象相比较。比较按照 字典序 进行： 首先比较最前面的两个元素， 如果不同， 就决定了比较的结果；如果相同， 就比较后面两个元素， 依此类推， 直到其中一个序列穷举完。如果要比较的两个元素本身就是同一类型的序列， 就按字典序递归比较。如果两个序列的所有元素都相等， 就为序认列相等。如果一个序列是另一个序列的初始子序列， 较短的序列就小于另一个。字符串的排序按照Unicode编码点的数值排序单个字符。下面是同类型序列之间比较的一些例子：

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

注意，使用< 或者 >比较不同类型的对象是合法的，只要这些对象具有合适的比较方法。例如，不同的数字类型按照它们的数值比较，所以 0 等于 0.0，等等。否则，解释器将引发一个TypeError异常，而不是随便给一个顺序。

脚注

[1]	**Other languages may return the mutated object, which allows method chaining, such as d->insert("a")->remove("b")->sort();.**

| [2] | Calling d.keys() will return a *dictionary view* object. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a *view*. | |-----|-----|

6. 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。因此，如果你想要编写一些更大的程序，最好使用文本编辑器先编写好，然后运行这个文件。这就是所谓的创建 脚本。随着你的程序变得越来越长，你可能想要将它分成几个文件，这样更易于维护。你还可能想在几个程序中使用你已经编写好的函数，而不用把函数拷贝到每个程序中。

为了支持这个功能，Python 有种方法可以把你定义的内容放到一个文件中，然后在脚本或者交互方式中使用。这种文件称为模块；模块中的定义可以 导入 到其它模块或 主模块 中。

模块是包含 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。在模块里面，模块的名字（是一个字符串）可以由全局变量 **name** 的值得到。例如，用你喜欢的文本编辑器在当前目录下创建一个名为 fibo.py 的文件，内容如下：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用下面的命令导入这个模块：

```
>>> import fibo
```

这不会直接把 fibo 中定义的函数的名字导入当前的符号表中；它只会把模块名字 fibo 导入其中。你可以通过模块名访问这些函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算频繁使用一个函数，可以将它赋给一个本地的变量：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. 深入模块

模块可以包含可执行语句以及函数的定义。这些语句通常用于初始化模块。它们只在 第一次导入时执行。[1]（如果文件以脚本的方式执行，它们也会运行。）

每个模块都有自己的私有符号表，模块内定义的所有函数用其作为全局符号表。因此，模块的作者可以在模块里使用全局变量，而不用担心与某个用户的全局变量有冲突。另一方面，如果你知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量，modname.itemname。

模块中可以导入其它模块。习惯上将所有的 import 语句放在模块（或者脚本）的开始，但这不是强制性的。被导入的模块的名字放在导入模块的全局符号表中。

import 语句的一个变体直接从被导入的模块中导入名字到模块的符号表中。例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这不会把模块名导入到本地的符号表中（所以在本例中，fibo 将没有定义）。

还有种方式可以导入模块中定义的所有名字：

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式不会导入以下划线 (_) 开头的名称。大多数情况下Python程序员不会使用这个便利的方法，因为它会引入一系列未知的名称到解释器中，这很可能隐藏你已经定义的一些东西。

注意一般情况下不赞成从一个模块或包中导入 *，因为这通常会导致代码很难读。不过，在交互式会话中这样用是可以的，它可以让你少敲一些代码。

注意

出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，你必需重新启动解释器 —— 或者，如果你就是想交互式的测试这么一个模块，可以使用imp.reload(), 例如 importimp;imp.reload(modulename)。

6.1.1. 执行模块

当你用下面的方式运行一个 Python 模块

```
python fibo.py <arguments>
```

模块中的代码将会被执行，就像导入它一样，不过此时 **name** 被设置为 **"main"** 。也就是说，如果你在模块后加入如下代码:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

就可以让此文件既可以作为可执行的脚本，也可以当作可以导入的模块，因为解析命令行的那部分代码只有在模块作为 “main” 文件执行时才被调用：

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

如果模块是被导入的，将不会运行这段代码：

```
>>> import fibo
>>>
```

这种方法通常用来为模块提供一个方便的用户接口，或者用来测试（例如直接运行脚本会执行一组测试用例）。

6.1.2. 模块搜索路径

当导入一个名为 spam 的模块时，解释器首先搜索具有该名称的内置模块。如果没有找到，它会接着到 sys.path 变量给出的目录中查找名为 spam.py 的文件。sys.path 变量的初始值来自这些位置：

- 脚本所在的目录（如果没有指明文件，则为当前目录）。
- PYTHONPATH （一个包含目录名的列表，与 shell 变量PATH 的语法相同）。
- 与安装相关的默认值。

Note

在支持符号连接的文件系统中，输入的脚本所在的目录是符号连接指向的目录。换句话说也就是包含符号链接的目录不会被加到目录搜索路径中。

初始化后，Python 程序可以修改sys.path。脚本所在的目录被放置在搜索路径的最开始，也就是在标准库的路径之前。这意味着将会加载当前目录中的脚本，库目录中具有相同名称的模块不会被加载。除非你是有意想替换标准库，否则这应该被当成是一个错误。更多信息请参阅 标准模块 小节。

6.1.3. "编译过的" Python 文件

为了加快加载模块的速度，Python会在**pycache**目录下以module.*version*.pyc名字缓存每个模块编译后的版本，这里的版本编制了编译后文件的格式。它通常会包含Python 的版本号。例如，在CPython 3.3版中，spam.py 编译后的版本将缓存为**pycache**/spam.cpython-33.pyc。这种命名约定允许由不同发布和不同版本的Python编译的模块同时存在。

Python会检查源文件与编译版的修改日期以确定它是否过期并需要重新编译。这是完全自动化的过程。同时，编译后的模块是跨平台的，所以同一个库可以在不同架构的系统之间共享。

Python 不检查在两个不同环境中的缓存。首先，它会永远重新编译而且不会存储直接从命令行加载的模块。其次，如果没有源模块它不会检查缓存。若要支持没有源文件（只有编译版）的发布，编译后的模块必须在源目录下，并且必须没有源文件的模块。

部分高级技巧：

- {{s.58}}{{s.59}}{{s.60}}{{s.61}}{{s.62}}

- {{s.63}}{{s.64}}{{s.65}}

- {{s.66}}{{s.67}}

- {{s.68}}{{s.69}}{{s.70}}

- {{s.71}}{{条例}}

- {{}} s.73

6.2. 标准模块

Python 带有一个标准模块库，并发布有单独的文档叫Python 库参考手册（以下简称"库参考手册"）。有些模块被直接构建在解析器里；这些操作虽然不是语言核心的部分，但是依然被内建进来，一方面是效率的原因，另一方面是为了提供访问操作系统原语如系统调用的功能。这些模块是可配置的，也取决于底层的平台。例如，winreg 模块只在 Windows 系统上提供。有一个特别的模块需要注意：sys，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和辅助提示符使用的字符串：

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

只有在交互式模式中，这两个变量才有定义。

变量 sys.path 是一个字符串列表，它决定了解释器搜索模块的路径。它初始的默认路径来自于环境变量 PYTHONPATH，如果 PYTHONPATH 未设置则来自于内置的默认值。你可以使用标准的列表操作修改它：

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. dir()函数

内置函数 dir() 用来找出模块中定义了哪些名字。它返回一个排好序的字符串列表：

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果不带参数， dir() 列出当前已定义的名称：

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

注意它列出了所有类型的名称： 变量、 模块、 函数等。

dir()不会列出内置的函数和变量的名称。如果你想列出这些内容，它们定义在标准模块 builtins 中：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4. 包

包是一种管理 Python 模块命名空间的方式，采用“点分模块名称”。例如，模块名 A.B 表示包 A 中一个名为 B 的子模块。就像模块的使用让不同模块的作者不用担心相互间的全局变量名称一样，点分模块的使用让包含多个模块的包（例如 Numpy 和 Python Imaging Library）的作者也不用担心相互之间的模块重名。

假设你想要设计一系列模块（或一个“包”）来统一处理声音文件和声音数据。现存很多种不同的声音文件格式 （通常由它们的扩展名来识别，例如： .wav, .aiff, .au）， 因此你可能需要创建和维护不断增长的模块集合来支持各种文件格式之间的转换。你可能还想针对音频数据做很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果）， 所以你还需要编写一组永远写不完的模块来处理这些操作。你的包可能会是这个结构（用分层的文件系统表示） ：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

导入这个包时，Python 搜索 sys.path 中的目录以寻找这个包的子目录。

为了让 Python 将目录当做包，目录下必须包含 **init**.py 文件；这样做是为了防止一个具有常见名字（例如 string）的目录无意中隐藏目录搜索路径中正确的模块。最简单的情况下，**init**.py 可以只是一个空的文件，但它也可以为包执行初始化代码或设置**all**变量（稍后会介绍）。

用户可以从包中导入单独的模块，例如：

```
import sound.effects.echo
```

这样就加载了子模块sound.effects.echo。它必须使用其完整名称来引用。

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

导入子模块的另一方法是：

```
from sound.effects import echo
```

这同样也加载了子模块echo，但使它可以不用包前缀访问，因此它可以按如下方式使用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有另一种变化方式是直接导入所需的函数或变量：

```
from sound.effects.echo import echofilter
```

这再次加载了子模块echo，但这种方式使函数echofilter() 可以直接访问：

```
echofilter(input, output, delay=0.7, atten=4)
```

注意使用frompackageimportitem时，item 可以是包的子模块（或子包），也可以是包中定义的一些其它的名称，比如函数、 类或者变量。import语句首先测试 item 在包中是否有定义；如果没有，它假定它是一个模块，并尝试加载它。如果未能找到，则引发ImportError异常。

相反，使用类似 importitem.subitem.subsubitem 这样的语法时，除了最后一项其它每项必须是一个包；最后一项可以是一个模块或一个包，但不能是在前一个项目中定义的类、函数或变量。

6.4.1. 从包中导入 *

那么现在如果用户写成 fromsound.effectsimport* 会发生什么？理想情况下，他应该是希望到文件系统中寻找包里面有哪些子模块，并把它们全部导入进来。这可能需要很长时间，而且导入子模块可能会产生想不到的副作用，这些作用本应该只有当子模块是显式导入时才会发生。

唯一的解决办法是包的作者为包提供显式的索引。import 语句使用以下约定： 如果包中的 **init**.py 代码定义了一个名为**all**的列表，那么在遇到 frompackageimport语句的时候，应该把这个列表中的所有模块名字导入。当包有新版本包发布时，就需要包的作者更新这个列表了。如果包的作者认为不可以用 *import* 方式导入它们的包，也可以决定不支持它。例如，文件sound/effects/**init**.py可以包含下面的代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这意味着 fromsound.effectsimport* 将导入sound 包的三个子模块。

如果 **all** 没有定义，fromsound.effectsimport 语句 不* 会从 sound.effects 包中导入所有的子模块到当前命名空间；它只保证 sound.effects 包已经被导入（可能会运行 **init**.py 中的任何初始化代码），然后导入包中定义的任何名称。这包括由 **init**.py 定义的任何名称（以及它显式加载的子模块）。还包括这个包中已经由前面的import 语句显式加载的子模块。请考虑此代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

在这个例子中，执行 from...import 语句时，echo 和 surround 模块被导入到当前命名空间是因为它们在sound.effects中有定义。（定义了 **all**时也会同样工作。）

虽然某些模块的设计旨在出口仅按照特定的模式，当您使用导入的名字，仍是不好练习在生产代码中。 虽然某些模块设计成使用 *import* 时只导出符合特定模式的名称，在产品代码中使用这种写法仍然是不好的做法。

记住，使用 fromPackageimportspecific_submodule 一点没错 ！事实上，这是推荐的写法，除非导入的模块需要使用其它包中的同名子模块。

6.4.2. 包内引用

如果一个包是子包（比如例子中的 sound 包），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块 sound.filters.vocoder 需要使用sound.effects 包中的 echo 模块，它可以使用 fromsound.effectsimportecho。

你还可以用frommoduleimportname形式的import语句进行相对导入。这些导入使用前导的点号表示相对导入的是从当前包还是上级的包。以 surround 模块为例，你可以使用：

```
from . import echo
from .. import formats
from ..filters import equalizer
```

注意，相对导入基于当前模块的名称。因为主模块的名字总是 "**main**"， Python 应用程序的主模块必须总是用绝对导入。

6.4.3. 包含多个目录的包

包还支持一个特殊的属性，**path**。该变量初始化一个包含 **init**.py 所在目录的列表。这个变量可以修改；这样做会影响未来包中包含的模块和子包的搜索。

虽然通常不需要此功能，它可以用于扩展包中的模块的集合。

脚注

[1]	**In fact function definitions are also ‘statements’ that are ‘executed’; the execution of a module-level function definition enters the function name in the module’s global symbol table.**

7. 输入和输出

展现程序的输出有多种方法；可以打印成人类可读的形式，也可以写入到文件以便后面使用。本章将讨论其中的几种方法。

7.1. 格式化输出

到目前为止我们遇到过两种输出值的方法：表达式语句和print()函数。（第三个方式是使用文件对象的write()方法；标准输出文件可以引用 sys.stdout。详细内容参见库参考手册。）

通常你会希望更好地控制输出的格式而不是简单地打印用空格分隔的值。有两种方法来设置输出格式；第一种方式是自己做所有的字符串处理；使用字符串切片和连接操作，你可以创建任何你能想象到的布局。字符串类型有一些方法，用于执行将字符串填充到指定列宽度的有用操作；这些稍后将讨论。第二种方法是使用str.format()方法。

string模块包含一个Template类，提供另外一种向字符串代入值得方法。

当然还有一个问题：如何将值转换为字符串？幸运的是，Python 有方法将任何值转换为字符串：将它传递给repr()或str()函数。

str()函数的用意在于返回人类可读的表现形式，而repr()的用意在于生成解释器可读的表现形式（如果没有等价的语法将会引发SyntaxError异常）。对于对人类并没有特别的表示形式的对象，str()和repr()将返回相同的值。许多值，例如数字或者列表和字典这样的结构，使用这两个函数中的任意一个都具有相同的表示形式。字符串，特殊一些，有两种不同的表示形式。

一些例子：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

这里用两种方法输出平方和立方表：

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

（注意在第一个示例中，每列之间的一个空格由print()自动添加：它总会在它的参数之间添加空格。）

上面的例子演示了字符串对象的str.rjust()方法，它通过在左侧填充空格使字符串在给定宽度的列右对齐。类似的方法还有str.ljust()和str.center()。这些方法不会输出任何内容，它们只返回新的字符串。如果输入的字符串太长，它们不会截断字符串，而是保持原样返回；这会使列的格式变得混乱，但是通常好于另外一种选择，那可能是一个错误的值。（如果你真的想要截断，可以加上一个切片操作，例如x.ljust(n)[:n]。）

另外一种方法 str.zfill()，它向数值字符串左侧填充零。该函数可以正确识别正负号：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format()方法的基本用法如下所示：

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

花括号及其中的字符（称为格式字段）将被替换为传递给str.format()方法的对象。可以用括号中的数字指定传递给str.format()方法的对象的位置。

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

如果str.format()方法使用关键字参数，那么将通过参数名引用它们的值。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置参数和关键字参数可以随意组合：

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

'!a'（用于ascii()），'!s'（用于str()）和'!r'（用于repr()）可以用来格式化之前对值进行转换。

```
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print('The value of PI is approximately {!r}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

字段名后允许可选的': '和格式指令。这允许更好地控制如何设置值的格式。下面的例子将 Pi 转为三位精度。

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

':'后面紧跟一个整数可以限定该字段的最小宽度。这在美化表格时很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

如果你有一个实在是很长的格式字符串但又不想分开写，要是可以按照名字而不是位置引用变量就好了。有个简单的方法，可以传入一个字典，然后使用'[]'访问。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这也可以用 '**' 标志将这个字典以关键字参数的方式传入。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这种方式与内置函数vars()组合起来将更加有用，该函数返回一个包含所有局部变量的字典。

关于str.format()完整的描述，请参见格式字符串语法。

7.1.1. 旧式的字符串格式

%运算符也可以用于字符串格式化。它将左边类似sprintf()-风格的参数应用到右边的参数，然后返回这种格式化操作生成的字符串。例如：

```
>>> import math
>>> print('The value of PI is approximately %5.3f.' % math.pi)
The value of PI is approximately 3.142.
```

在*printf-style String Formatting*一节，可以找到更多的信息。

7.2. 读写文件

open()返回一个文件对象，最常见的用法带有两个参数：open(filename,mode)。

```
>>> f = open('workfile', 'w')
```

第一个参数是一个含有文件名的字符串。第二个参数也是一个字符串，含有描述如何使用该文件的几个字符。*mode*为'r'时表示只是读取文件；w 表示只是写入文件（已经存在的同名文件将被删掉）；'a'表示打开文件进行追加，写入到文件中的任何数据将自动添加到末尾。'r+'表示打开文件进行读取和写入。*mode* 参数是可选的，默认为'r'。

通常，文件以文本打开，这意味着，你从文件读出和向文件写入的字符串会被特定的编码方式（默认是UTF-8）编码。模式后面的'b'以二进制模式打开文件：数据会以字节对象的形式读出和写入。这种模式应该用于所有不包含文本的文件。

在文本模式下，读取时默认会将平台有关的行结束符（Unix上是\n, Windows上是\r\n）转换为\n。在文本模式下写入时，默认会将出现的\n转换成平台有关的行结束符。这种暗地里的修改对 ASCII 文本文件没有问题，但会损坏JPEG或EXE这样的二进制文件中的数据。使用二进制模式读写此类文件时要特别小心。

7.2.1. 文件对象的方法

本节中的示例将假设文件对象f已经创建。

要读取文件内容，可以调用f.read(size)，该方法读取若干数量的数据并以字符串或字节对象返回。*size* 是可选的数值参数。当 *size* 被省略或者为负数时，将会读取并返回整个文件；如果文件大小是你机器内存的两倍时，就是你的问题了。否则，至多读取和返回 *size* 大小的字节数据。如果到了文件末尾，f.read() 会返回一个空字符串('')。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline()从文件读取一行数据；字符串结尾会带有一个换行符 (\n)，只有当文件最后一行没有以换行符结尾时才会省略。这样返回值就不会有混淆，如果 f.readline()返回一个空字符串，那就表示已经达到文件的末尾，而如果返回一个只包含一个换行符的字符串'\n'，则表示遇到一个空行。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

你可以循环遍历文件对象来读取文件中的每一行。这是既省内存又非常快的简单代码：

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

如果你想把文件中的所有行读到一个列表中，你也可以使用list(f)或f.readlines()。

f.write(string)将 *string* 的内容写入文件中并返回写入的字节的数目。

```
>>> f.write('This is a test\n')
15
```

如果想写入字符串以外的数据，需要先将它转换为一个字符串：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

f.tell()返回一个给出文件对象在文件中当前位置的整数，在二进制模式下表示自文件开头的比特数，在文本模式下是一个无法理解的数。

若要更改该文件对象的位置，可以使用f.seek(offset,from_what)。新的位置由参考点加上 *offse* 计算得来，参考点的选择则来自于 *from_what* 参数。*from_what* 值为 0 表示以文件的开始为参考点，1 表示以当前的文件位置为参考点，2 表示以文件的结尾为参考点。*from_what* 可以省略，默认值为 0，表示以文件的开始作为参考点。

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

在文本文件中（没有以b模式打开），只允许从文件头开始寻找（有个例外是用seek(0,2)寻找文件的最末尾处）而且合法的偏移值只能是f.tell()返回的值或者是零。其它任何偏移 值都会产生未定义的行为。

使用完一个文件后，调用f.close()可以关闭它并释放其占用的所有系统资源。调用f.close()后，再尝试使用该文件对象将失败。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

处理文件对象时使用with关键字是很好的做法。这样做的好处在于文件用完后会自动关闭，即使过程中发生异常也没关系。它还比编写一个等同的try-finally语句要短很多：

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有一些不太常用的方法，例如isatty()和truncate()；有关文件对象的完整指南，请参阅 Python 库参考手册。

7.2.2. 使用json存储结构化数据

从文件中读写字符串很容易。数值就要多费点儿周折，因为read ()方法只会返回字符串，应将其传入int()这样的函数，就可以将'123'这样的字符串转换为对应的数值 123。当你想要保存更为复杂的数据类型，例如嵌套的列表和字典，手工解析和序列化它们将变得更复杂。

好在用户不是非得自己编写和调试保存复杂数据类型的代码，Python 允许你使用常用的数据交换格式JSON（JavaScript Object Notation）。标准模块json可以接受 Python 数据结构，并将它们转换为字符串表示形式；此过程称为序列化。从字符串表示形式重新构建数据结构称为反序列化。序列化和反序列化的过程中，表示该对象的字符串可以存储在文件或数据中，也可以通过网络连接传送给远程的机器。

注意

JSON 格式经常用于现代应用程序中进行数据交换。许多程序员都已经熟悉它了，使它成为相互协作的一个不错的选择。

如果你有一个对象x，你可以用简单的一行代码查看其 JSON 字符串表示形式：

```
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

dumps()函数的另外一个变体dump()，直接将对象序列化到一个文本文件。所以如果f是为写入而打开的一个文本文件对象，我们可以这样做：

```
json.dump(x, f)
```

为了重新解码对象，如果f是为读取而打开的文本文件对象：

```
x = json.load(f)
```

这种简单的序列化技术可以处理列表和字典，但序列化任意类实例为 JSON 需要一点额外的努力。Json模块的手册对此有详细的解释。

另请参阅

pickle - pickle模块

与*JSON* 不同，*pickle* 是一个协议，它允许任意复杂的 Python 对象的序列化。因此，它只能用于 Python 而不能用来与其他语言编写的应用程序进行通信。默认情况下它也是不安全的：如果数据由熟练的攻击者精心设计， 反序列化来自一个不受信任源的 pickle 数据可以执行任意代码。

8. 错误和异常

直到现在，我们还没有更多的提及错误信息，但是如果你真的尝试了前面的例子，也许你已经见到过一些。Python（至少）有两种错误很容易区分：语法错误 和异常。

8.1. 语法错误

语法错误，或者称之为解析错误，可能是你在学习 Python 过程中最烦的一种：

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的'箭头'。错误是由箭头前面 的标记引起的（至少检测到是这样的）： 在这个例子中，检测到错误发生在函数print()，因为在它之前缺少一个冒号（':'）。文件名和行号会一并输出，所以如果运行的是一个脚本你就知道去哪里检查错误了。

8.2. 异常

即使一条语句或表达式在语法上是正确的，在运行它的时候，也有可能发生错误。在执行期间检测到的错误被称为异常 并且程序不会无条件地崩溃：你很快就会知道如何在 Python 程序中处理它们。然而大多数异常都不会被程序处理，导致产生类似下面的错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

最后一行的错误消息指示发生了什么事。异常有不同的类型，其类型会作为消息的一部分打印出来：在这个例子中的类型有ZeroDivisionError、 NameError和TypeError。打印出来的异常类型的字符串就是内置的异常的名称。这对于所有内置的异常是正确的，但是对于用户自

定义的异常就不一定了（尽管这是非常有用的惯例）。标准异常的名称都是内置的标识符（不是保留的关键字）。

这一行最后一部分给出了异常的详细信息和引起异常的原因。

错误信息的前面部分以堆栈回溯的形式显示了异常发生的上下文。通常调用栈里会包含源代码的行信息，但是来自标准输入的源码不会显示行信息。

内置的异常 列出了内置的异常以及它们的含义。

8.3. 抛出异常

可以通过编程来选择处理部分异常。看一下下面的例子，它会一直要求用户输入直到输入一个合法的整数为止，但允许用户中断这个程序（使用Control-C或系统支持的任何方法）；注意用户产生的中断引发的是 KeyboardInterrupt 异常。

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

Try语句按以下方式工作。

- 首先，执行*try* 子句（try和except关键字之间的语句）。
- 如果未发生任何异常，忽略*except* 子句且try语句执行完毕。
- 如果在 try 子句执行过程中发生异常，跳过该子句的其余部分。如果异常的类型与except 关键字后面的异常名匹配, 则执行 except 子句，然后继续执行try语句之后的代码。
- 如果异常的类型与 except 关键字后面的异常名不匹配，它将被传递给上层的try语句；如果没有找到处理这个异常的代码，它就成为一个未处理异常，程序会终止运行并显示一条如上所示的信息。

Try语句可能有多个子句，以指定不同的异常处理程序。不过至多只有一个处理程序将被执行。处理程序只处理发生在相应 try 子句中的异常，不会处理同一个try字句的其他处理程序中发生的异常。一个 except 子句可以用带括号的元组列出多个异常的名字，例如：

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

最后一个 except 子句可以省略异常名称，以当作通配符使用。使用这种方式要特别小心，因为它会隐藏一个真实的程序错误！它还可以用来打印一条错误消息，然后重新引发异常 （让调用者也去处理这个异常）：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try...except语句有一个可选的*else* 子句，其出现时，必须放在所有 except 子句的后面。如果需要在 try 语句没有抛出异常时执行一些代码，可以使用这个子句。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else子句比把额外的代码放在try子句中要好，因为它可以避免意外捕获不是由try ...保护的代码所引发的异常。除了语句。

当异常发生时，它可能带有相关数据，也称为异常的参数。参数的有无和类型取决于异常的类型。

except 子句可以在异常名之后指定一个变量。这个变量将绑定于一个异常实例，同时异常的参数将存放在实例的args中。为方便起见，异常实例定义了**str()**，因此异常的参数可以直接打印而不必引用.args。也可以在引发异常之前先实例化一个异常，然后向它添加任何想要的属性。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于未处理的异常，如果它含有参数，那么参数会作为异常信息的最后一部分打印出来。

异常处理程序不仅处理直接发生在 try 子句中的异常，而且还处理 try 子句中调用的函数（甚至间接调用的函数）引发的异常。例如：

```
>>> def this_fails():
...      x = 1/0
...
>>> try:
...      this_fails()
... except ZeroDivisionError as err:
...      print('Handling run-time error:', err)
...
Handling run-time error: int division or modulo by zero
```

8.4. 引发异常

raise语句允许程序员强行引发一个指定的异常。例如：

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise的唯一参数指示要引发的异常。它必须是一个异常实例或异常类（从Exception派生的类）。

如果你确定需要引发异常，但不打算处理它，一个简单形式的raise语句允许你重新引发异常：

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5. 用户定义的异常

程序可以通过创建新的异常类来命名自己的异常（Python 类的更多内容请参见类）。异常通常应该继承Exception类，直接继承或者间接继承都可以。例如：

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在此示例中，Exception默认的**init()**被覆盖了。新的行为简单地创建了*value* 属性。这将替换默认的创建*args* 属性的行为。

异常类可以像其他类一样做任何事情，但是通常都会比较简单，只提供一些属性以允许异常处理程序获取错误相关的信息。创建一个能够引发几种不同错误的模块时，一个通常的做法是为该模块定义的异常创建一个基类，然后基于这个基类为不同的错误情况创建特定的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

大多数异常的名字都以"Error"结尾，类似于标准异常的命名。

很多标准模块中都定义了自己的异常来报告在它们所定义的函数中可能发生的错误。类 这一章给出了类的详细信息。

8.6. 定义清理操作

Try语句有另一个可选的子句，目的在于定义必须在所有情况下执行的清理操作。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
```

不管有没有发生异常，在离开try语句之前总是会执行*finally*子句。当try子句中发生了一个异常，并且没有except字句处理（或者异常发生在try或else子句中），在执行完finally子句后将重新引发这个异常。try语句由于break、contine或return语句离开时，同样会执行finally子句。下面是一个更复杂些的例子：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

正如您所看到的，在任何情况下都会执行finally子句。由两个字符串相除引发的 TypeError异常没有被except子句处理，因此在执行finally子句后被重新引发。

在真实的应用程序中， finally子句用于释放外部资源（例如文件或网络连接），不管资源的使用是否成功。

8.7. 清理操作的预定义

有些对象定义了在不需要该对象时的标准清理操作，无论该对象的使用是成功还是失败。看看下面的示例，它尝试打开一个文件并打印其内容到屏幕。

```
for line in open("myfile.txt"):
    print(line, end="")
```

这段代码的问题就是这部分代码执行完之后它还会让文件在一段不确定的时间内保持打开状态。这在简单的脚本中没什么，但是在大型应用程序中可能是一个问题。With语句可以确保像文件这样的对象总能及时准确地被清理掉。

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

执行该语句后，文件*f* 将始终被关闭，即使在处理某一行时遇到了问题。提供预定义的清理行为的对象，和文件一样，会在它们的文档里说明。

9. 类

与其他编程语言相比，Python 的类机制用最少的语法和语义引入了类。它是 C++ 和 Modula-3 类机制的混合。Python 的类提供了面向对象编程的所有标准功能： 类继承机制允许有多个基类， 继承的类可以覆盖其基类或类的任何方法，方法能够以相同的名称调用基类中的方法。对象可以包含任意数量和种类的数据。和模块一样， 类同样具有 Python 的动态性质：它们在运行时创建，并可以在创建之后进一步修改。

用 C++ 术语来讲，通常情况下类成员（包括数据成员）是公有的（其它情况见下文私有变量），所有的成员函数都是虚 的。与 Modula-3 一样，在成员方法中没有简便的方式引用对象的成员：方法函数的声明用显式的第一个参数表示对象本身，调用时会隐式地引用该对象。与 Smalltalk 一样， 类本身也是对象。这给导入类和重命名类提供了语义上的合理性。与 C++ 和 Modula-3 不同，用户可以用内置类型作为基类进行扩展。此外，像 C++ 一样， 类实例可以重定义大多数带有特殊语法的内置操作符（算术运算符、 下标等）。

（由于没有统一的达成共识的术语，我会偶尔使用 SmallTalk 和 C++ 的术语。我比较喜欢用 Modula-3 的术语，因为比起 C++，Python 的面向对象语法更像它，但是我想很少有读者听说过它。）

9.1. 名称和对象

对象是独立的，多个名字（在多个作用域中）可以绑定到同一个对象。这在其他语言中称为别名。第一次粗略浏览 Python 时经常不会注意到这个特性，而且处理不可变的基本类型（数字，字符串，元组）时忽略这一点也没什么问题。然而， 在Python 代码涉及可变对象如列表、 字典和大多数其它类型时，别名可能具有意想不到语义效果。这通常有助于优化程序，因为别名的行为在某些方面类似指针。例如，传递一个对象的开销是很小的，因为在实现上只是传递了一个指针；如果函数修改了参数传递的对象，调用者也将看到变化 —— 这就避免了类似 Pascal 中需要两个不同参数的传递机制。

9.2. Python 作用域和命名空间

在介绍类之前，首先我要告诉你一些有关 Python 作用域的的规则。类的定义非常巧妙的运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的工作原理。另外，这一切的知识对于任何高级 Python 程序员都非常有用。

让我们从一些定义开始。

命名空间是从名称到对象的映射。当前命名空间主要是通过 Python 字典实现的，不过通常不会引起任何关注（除了性能方面），它以后也有可能会改变。以下有一些命名空间的例子：内置名称集（包括函数名列如abs()和内置异常的名称）；模块中的全局名称；函数调用中的

局部名称。在某种意义上的一组对象的属性也形成一个命名空间。关于命名空间需要知道的重要一点是不同命名空间的名称绝对没有任何关系；例如，两个不同模块可以都定义函数maximize而不会产生混淆 —— 模块的使用者必须以模块名为前缀引用它们。

顺便说一句，我使用属性 这个词称呼点后面的任何名称 —— 例如，在表达式z.real中，real是z对象的一个属性。严格地说，对模块中的名称的引用是属性引用：在表达式modname.funcname中， modname是一个模块对象，funcname是它的一个属性。在这种情况下，模块的属性和模块中定义的全局名称之间碰巧是直接的映射：它们共享同一命名空间！[1]

属性可以是只读的也可以是可写的。在后一种情况下，可以对属性赋值。模块的属性都是可写的：你可以这样写modname.the_answer=42。可写的属性也可以用del语句删除。例如，delmodname.the_answer将会删除对象modname中的the_answer属性。

各个命名空间创建的时刻是不一样的，且有着不同的生命周期。包含内置名称的命名空间在Python 解释器启动时创建，永远不会被删除。模块的全局命名空间在读入模块定义时创建；通常情况下，模块命名空间也会一直保存到解释器退出。在解释器最外层调用执行的语句，不管是从脚本文件中读入还是来自交互式输入，都被当作模块**main**的一部分，所以它们有它们自己的全局命名空间。（内置名称实际上也存在于一个模块中，这个模块叫builtins。）

函数的局部命名空间在函数调用时创建，在函数返回或者引发了一个函数内部没有处理的异常时删除。（实际上，用遗忘来形容到底发生了什么更为贴切。)当然，每个递归调用有它们自己的局部命名空间。

作用域 是 Python 程序中可以直接访问一个命名空间的代码区域。这里的“直接访问”的意思是用没有前缀的引用在命名空间中找到的相应的名称。

虽然作用域的确定是静态地，但它们的使用是动态地。程序执行过程中的任何时候，至少有三个嵌套的作用域，它们的命名空间是可以直接访问的：

- 首先搜索最里面包含局部命名的作用域
- 其次搜索所有调用函数的作用域，从最内层调用函数的作用域开始，它们包含非局部但也非全局的命名
- 倒数第二个搜索的作用域是包含当前模块全局命名的作用域
- 最后搜索的作用域是最外面包含内置命名的命名空间

如果一个命名声明为全局的，那么对它的所有引用和赋值会直接搜索包含这个模块全局命名的作用域。如果要重新绑定最里层作用域之外的变量，可以使用nonlocal语句；如果不声明为nonlocal，这些变量将是只读的（对这样的变量赋值会在最里面的作用域创建一个新 的局部变量，外部具有相同命名的那个变量不会改变）。

通常情况下，局部作用域引用当前函数的本地命名。函数之外，局部作用域引用的命名空间与全局作用域相同：模块的命名空间。类定义在局部命名空间中创建了另一个命名空间。

认识到作用域是由代码确定的是非常重要的：函数的全局作用域是函数的定义所在的模块的命名空间，与函数调用的位置或者别名无关。另一方面，命名的实际搜索过程是动态的，在运行时确定的——然而，Python 语言也在不断发展，以后有可能会成为静态的“编译”时确定，所以不要依赖动态解析！（事实上，本地变量是已经确定静态。）

Python的一个特别之处在于——如果没有使用global语法——其赋值操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除也是如此：delx只是从局部作用域的命名空间中删除命名x。事实上，所有引入新命名的操作都作用于局部作用域： 特别是import语句和函数定将模块名或函数绑定于局部作用域。(可以使用 Global 语句将变量引入到全局作用域。)

global语句可以用来指明某个特定的表明位于全局作用域并且应该在那里重新绑定；nonlocal语句表示特定的变量位于一个封闭的作用域并且应该在那里重新绑定。

9.2.1. 作用域和命名空间示例

下面这个示例演示如何访问不同作用域和命名空间，以及global 和 nonlocal 如何影响变量的绑定:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

示例代码的输出为：

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spams.
In global scope: global spam
```

注意，*local*赋值（默认行为）没有改变*scope_test**spam*的绑定。nonlocal赋值改变了*scope_test*对*spam* 的绑定，global赋值改变了模块级别的绑定。

你也可以看到在global语句之前没有对*spam*的绑定。

9.3. 初识类

类引入了少量的新语法、三种新对象类型和一些新语义。

9.3.1. 类定义语法

类定义的最简单形式如下所示：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类的定义就像函数定义（def语句），要先执行才能生效。(你当然可以把它放进if语句的某一分支，或者一个函数的内部。)

实际应用中，类定义包含的语句通常是函数定义，不过其它语句也是可以的而且有时还会很有用——后面我们会再回来讨论。类中的函数定义通常有一个特殊形式的参数列表，这是由方法调用的协议决定的——同样后面会解释这些。

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有的赋值成为这个新命名空间的局部变量。特别是这里的函数定义会绑定新函数的名字。

类定义正常退出时，一个类对象也就创建了。基本上它是对类定义创建的命名空间进行了一个包装；我们在下一节将进一步学习类对象的知识。原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名（例子中是ClassName）。

9.3.2. 类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用的所有属性引用在 Python 中使用的标准语法： obj.name。有效的属性名称是在该类的命名空间中的类对象被创建时的所有名称。因此，如果类定义看起来像这样：

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

那么 MyClass.i 和 MyClass.f 是有效的属性引用，分别返回一个整数和一个方法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。**doc** 也是一个有效的属性，返回类的文档字符串： "Asimpleexampleclass"。

类的实例化 使用函数的符号。可以假设类对象是一个不带参数的函数，该函数返回这个类的一个新的实例。例如（假设沿用上面的类）：

```
x = MyClass()
```

创建这个类的一个新实例，并将该对象赋给局部变量x。

实例化操作（“调用”一个类对象）将创建一个空对象。很多类希望创建的对象可以自定义一个初始状态。因此类可以定义一个名为**init()**的特殊方法，像下面这样：

```
def __init__(self):
    self.data = []
```

当类定义了**init()**方法，类的实例化会为新创建的类实例自动调用**init()**。所以在下面的示例中，可以获得一个新的、已初始化的实例：

```
x = MyClass()
```

当然，**init()**方法可以带有参数，这将带来更大的灵活性。在这种情况下，类实例化操作的参数将传递给**init()**。例如，

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. 实例对象

现在我们可以用实例对象做什么？实例对象唯一可用的操作就是属性引用。有两种有效的属性名：数据属性和方法。

数据属性相当于 Smalltalk 中的"实例变量"或 C++ 中的"数据成员"。数据属性不需要声明；和局部变量一样，它们会在第一次给它们赋值时生成。例如，如果x是上面创建的MyClass的实例，下面的代码段将打印出值16而不会出现错误：

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

实例属性引用的另一种类型是方法。方法是"属于"一个对象的函数。（在 Python，方法这个术语不只针对类实例：其他对象类型也可以具有方法。例如，列表对象有 append、insert、remove、sort 方法等等。但是在后面的讨论中，除非明确说明，我们提到的方法特指类实例对象的方法。）

实例对象的方法的有效名称依赖于它的类。根据定义，类中所有函数对象的属性定义了其实例中相应的方法。所以在我们的示例中， x.f是一个有效的方法的引用，因为MyClass.f是一个函数，但x.i不是，因为MyClass.i不是一个函数。但x.f与MyClass.f也不是一回事 —— 它是一个方法对象，不是一个函数对象。

9.3.4. 方法对象

通常情况下，方法在绑定之后被直接调用：

```
x.f()
```

在MyClass的示例中，这将返回字符串'helloworld'。然而，也不是一定要直接调用方法： x.f 是一个方法对象，可以存储起来以后调用。例如：

```
xf = x.f
while True:
    print(xf())
```

会不断地打印helloworld。

调用方法时到底发生了什么？你可能已经注意到，上面x.f()的调用没有参数，即使f ()函数的定义指定了一个参数。该参数发生了什么问题？当然如果函数调用中缺少参数 Python 会抛出异常——即使这个参数实际上没有使用……

实际上，你可能已经猜到了答案：方法的特别之处在于实例对象被作为函数的第一个参数传给了函数。在我们的示例中，调用x.f()完全等同于MyClass.f(x)。一般情况下，以*n* 个参数的列表调用一个方法就相当于将方法所属的对象插入到列表的第一个参数的前面，然后以新的列表调用相应的函数。

如果你还是不明白方法的工作原理，了解一下它的实现或许有帮助。引用非数据属性的实例属性时，会搜索它的类。如果这个命名确认为一个有效的函数对象类属性，就会将实例对象和函数对象封装进一个抽象对象：这就是方法对象。以一个参数列表调用方法对象时，它被重新拆封，用实例对象和原始的参数列表构造一个新的参数列表，然后函数对象调用这个新的参数列表。

9.3.5. 类和实例变量

一般来说，实例变量用于对每一个实例都是唯一的数据，类变量用于类的所有实例共享的属性和方法：

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

正如在名称和对象讨论的，可变对象，例如列表和字典，的共享数据可能带来意外的效果。例如，下面代码中的*tricks* 列表不应该用作类变量，因为所有的*Dog* 实例将共享同一个列表：

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

这个类的正确设计应该使用一个实例变量：

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. 补充说明

数据属性会覆盖同名的方法属性；为了避免意外的命名冲突，这在大型程序中可能带来极难发现的 bug，使用一些约定来减少冲突的机会是明智的。可能的约定包括大写方法名称的首字母，使用一个唯一的小写的字符串（也许只是一个下划线）作为数据属性名称的前缀，或者方法使用动词而数据属性使用名词。

数据属性可以被方法引用，也可以由一个对象的普通用户（“客户端”）使用。换句话说，类是不能用来实现纯抽象数据类型。事实上，Python 中不可能强制隐藏数据——一切基于约定。（另一方面，如果需要，使用 C 编写的 Python 实现可以完全隐藏实现细节并控制对象的访

问；这可以用来通过 C 语言扩展 Python。）

客户应该谨慎的使用数据属性——客户可能通过践踏他们的数据属性而使那些由方法维护的常量变得混乱。注意：只要能避免冲突，客户可以向一个实例对象添加他们自己的数据属性，而不会影响方法的正确性——再次强调，命名约定可以避免很多麻烦。

从方法内部引用数据属性（或其他方法）并没有快捷方式。我觉得这实际上增加了方法的可读性：当浏览一个方法时，在局部变量和实例变量之间不会出现令人费解的情况。

通常，方法的第一个参数称为self。这仅仅是一个约定：名字self对 Python 而言绝对没有任何特殊含义。但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些类 查看 器程序也可能是遵循此约定编写的。

类属性的任何函数对象都为那个类的实例定义了一个方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在f、 g和h都是类C中引用函数对象的属性，因此它们都是c的实例的方法 —— h完全等同于g。请注意，这种做法通常只会混淆程序的读者。

通过使用self利用参数的方法属性，可以调用其他方法：

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以像普通函数那样引用全局命名。与方法关联的全局作用域是包含类定义的模块。（类本身永远不会做为全局作用域使用。）尽管很少有好的理由在方法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用导入全局作用域的函数和方法，也可以调用定义在其中的类和函数。通常，包含此方法的类也会定义在这个全局作用域，在下一节我们会了解为何一个方法要引用自己的类。

每个值是都一个对象，因此每个值都有一个类（也称它的类型）。它存储为object.**class**。

9.5. 继承

当然，一个语言特性不支持继承是配不上“类”这个名字的。派生类定义的语法如下所示：

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

BaseClassName必须与派生类定义在一个作用域内。用其他任意表达式代替基类的名称也是允许的。这可以是有用的，例如，当基类定义在另一个模块中时：

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程和基类是相同的。类对象创建后，基类会被保存。这用于解析属性的引用：如果在类中找不到请求的属性，搜索会在基类中继续。如果基类本身是由别的类派生而来，这个规则会递归应用。

派生类的实例化没有什么特殊之处：DerivedClassName()创建类的一个新的实例。方法的引用按如下规则解析： 搜索对应的类的属性，必要时沿基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

派生的类可能重写其基类的方法。因为方法调用同一个对象中的其它方法时没有特权，基类的方法调用同一个基类的方法时，可能实际上最终调用了派生类中的覆盖方法。（对于 C++ 程序员：Python 中的所有方法实际上都是虚的。）

派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一个简单的方法可以直接调用基类方法：只要调用BaseClassName.methodname(self,arguments)。有时这对于客户端也很有用。（要注意只有BaseClassName在同一全局作用域定义或导入时才能这样用。）

Python 有两个用于继承的函数：

- 使用isinstance()来检查实例类型：isinstance(obj, int)只有obj.**class**是int或者是从int派生的类时才为True。
- 使用issubclass()来检查类的继承: issubclas（bool，int)是True因为bool是int的子类。然而，issubclass(float,int)为False，因为float不是int的子类。

9.5.1. 多继承

Python 也支持一种形式的多继承。具有多个基类的类定义如下所示：

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

对于大多数用途，在最简单的情况下，你可以认为继承自父类的属性搜索是从左到右的深度优先搜索，不会在同一个类中搜索两次，即使层次会有重叠。因此，如果在DerivedClassName中找不到属性，它搜索Base1，然后（递归）基类中的Base1，如果没有找到，它会搜索base2，依此类推。

事实要稍微复杂一些；为了支持合作调用super()，方法解析的顺序会动态改变。这种方法在某些其它多继承的语言中也有并叫做call-next-method，它比单继承语言中的super调用更强大。

动态调整顺序是必要的，因为所有的多继承都会有一个或多个菱形关系(从最底部的类向上，至少会有一个父类可以通过多条路径访问到）。例如，所有的类都继承自object，所以任何多继承都会有多条路径到达object。为了防止基类被重复访问，动态算法线性化搜索顺序，每个类都按从左到右的顺序特别指定了顺序，每个父类只调用一次，这是单调的（也就是说一个类被继承时不会影响它祖先的次序）。所有这些特性使得设计可靠并且可扩展的多继承类成为可能。有关详细信息，请参阅http://www.python.org/download/releases/2.3/mro/。

9.6. 私有变量

在 Python 中不存在只能从对像内部访问的“私有”实例变量。然而，有一项大多数 Python 代码都遵循的公约：带有下划线（例如_spam）前缀的名称应被视为非公开的 API 的一部分（无论是函数、 方法还是数据成员）。它应该被当做一个实现细节，将来如果有变化恕不另行通知。

因为有一个合理的类私有成员的使用场景（即为了避免名称与子类定义的名称冲突），Python 对这种机制有简单的支持，叫做*name mangling*。**spam** 形式的任何标识符(前面至少两个下划线，后面至多一个下划线）将被替换为**_classname**spam，classname是当前类的名字。此重整是做而不考虑该标识符的句法位置，只要它出现在类的定义的范围内。

Name mangling 有利于子类重写父类的方法而不会破坏类内部的方法调用。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

请注意 mangling 规则的目的主要是避免发生意外；访问或者修改私有变量仍然是可能的。这在特殊情况下，例如调试的时候，还是有用的。

注意传递给exec或eval()的代码没有考虑要将调用类的类名当作当前类；这类似于global语句的效果，影响只限于一起进行字节编译的代码。相同的限制适用于getattr()、 setattr()和delattr()，以及直接引用**dict**时。

9.7. 零碎的说明

有时候类似于Pascal 的"record" 或 C 的"struct"的数据类型很有用，它们把几个已命名的数据项目绑定在一起。一个空的类定义可以很好地做到：

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

某一段 Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类来模拟该数据类型的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有read ()和readline () 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

实例的方法对象也有属性：m.**self**是具有方法m()的实例对象，m.**func**是方法的函数对象。

9.8. 异常也是类

用户定义的异常类也由类标识。利用这个机制可以创建可扩展的异常层次。

raise语句有两种新的有效的（语义上的）形式：

```
raise Class

raise Instance
```

第一种形式中，Class 必须是type或者它的子类的一个实例。第一种形式是一种简写：

```
raise Class()
```

except子句中的类如果与异常是同一个类或者是其基类，那么它们就是相容的（但是反过来是不行的——except子句列出的子类与基类是不相容的）。例如，下面的代码将按该顺序打印 B、 C、 D：

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

请注意，如果except 子句的顺序倒过来 (excpetB在最前面），它就会打印B，B，B —— 第一个匹配的异常被触发。

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后是用内置函数str()将类转换得到的完整字符串。

9.9. 迭代器

现在你可能注意到大多数容器对象都可以用for遍历：

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

这种访问风格清晰、 简洁又方便。迭代器的用法在 Python 中普遍而且统一。在后台， for语句调用容器对象上的iter() 。该函数返回一个定义了**next()**方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，**next()**会引发StopIteration异常，告诉for循环终止。你可以是用内建的next()函数调用**next()**方法；此示例显示它是如何工作：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

看过迭代器协议背后的机制后，将很容易将迭代器的行为添加到你的类中。定义一个**iter()**方法，它使用**next()**方法返回一个对象。如果类定义了**next()**，**iter()**可以只返回self：

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.10. 生成器

生成器是简单且功能强大的工具，用于创建迭代器。它们写起来就像是正规的函数，需要返回数据的时候使用yield语句。每次在它上面调用next()时，生成器恢复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了生成器可以非常简单地创建出来：

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

生成器能做到的什么事，前一节所述的基于类的迭代器也能做到。生成器这么简洁是因为**iter()**和**next()**方法是自动创建的。

另一个关键特征是在调用中本地变量和执行状态会自动保存，这使得该函数更容易易写，也比使用实例变量的方法，如self.index和self.data更清晰。

除了自动创建方法和保存程序的状态，当生成器终止时，它们会自动引发StopIteration。在组合中，这些功能可以容易地创建迭代器而不需要写正则函数

9.11. 生成器表达式

一些简单的生成器可以简洁地使用表达式，语法类似于列表格式，但用圆括号()而不是方括号[]。这些表达式用于闭包函数马上使用生成器的情况。生成器表达式更紧凑但比完整生成器定义较不通用，倾向于更多的内存友好比等效列表中体会。

例子：

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

脚注

| [1] | Except for one thing. Module objects have a secret read-only attribute called **dict** which returns the dictionary used to implement the module's namespace; the name **dict** is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers. | |-----|-----|

10. 标准库概览

10.1. 操作系统接口

os模块提供了几十个函数与操作系统交互：

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python34'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

一定要使用importos的形式而不要用fromosimport*。这将避免os.open()屏蔽内置的open()函数，它们的功能完全不同。

内置的dir()和help()函数对于使用像os大型模块可以作为非常有用的交互式帮助：

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

对于日常的文件和目录管理任务，shutil模块提供了一个易于使用的高级接口：

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2. 文件通配符

glob模块提供了一个函数用于在目录中以通配符搜索文件，并生成匹配的文件列表：

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. 命令行参数

常见的实用程序脚本通常需要处理命令行参数。这些参数以一个列表存储在sys模块的*argv* 属性中。例如下面的输出结果来自于从命令行运行pythondemo.pyone two three：

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

getopt模块使用Unix getopt()函数的约定处理*sys.argv*。argparse模块提供更强大、 更灵活的命令行处理功能。

10.4. 错误输出重定向和程序终止

sys模块还具有*stdin*、*stdout*和*stderr*属性。即使在*stdout*被重定向时，后者也可以用于显示警告和错误信息：

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

终止脚本最直接的方法来是使用sys.exit()。

10.5. 字符串模式匹配

re模块为高级的字符串处理提供了正则表达式工具。对于复杂的匹配和操作，正则表达式提供了简洁、优化的解决方案：

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

当只需要简单的功能时，最好使用字符串方法，因为它们更容易阅读和调试：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. 数学

math模块为浮点运算提供了对底层 C 函数库的访问：

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random模块提供了进行随机选择的工具：

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

SciPy项目http://scipy.org有很多其它用于数值计算的模块。

10.7. 互联网访问

有很多的模块用于访问互联网和处理的互联网协议。最简单的两个是从URL获取数据的urllib.request 和发送邮件的smtplib：

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')  # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

（请注意第二个示例需要在本地主机上运行邮件服务器）。

10.8. 日期和时间

datetime模块提供了处理日期和时间的类，既有简单的方法也有复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持处理时区。

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. 数据压缩

常见的数据打包和压缩格式有模块直接支持，包括：zlib, gzip, bz2, lzma, zipfile 和 tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. 性能度量

一些 Python 用户对同一问题的不同解决方法之间的性能差异深有兴趣。Python 提供了的一个度量工具可以立即解决这些问题。

例如，使用元组封装和拆封功能而不是传统的方法来交换参数可能会更吸引人。timeit模块快速证明了现代的方法更快一些：

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

与timeit的精细的粒度相反，profile和pstats模块提供了针对更大代码块的时间度量工具。

10.11. 质量控制

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。

doctest模块提供一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。通过用户提供的例子，它发展了文档，允许 doctest 模块确认代码的结果是否与文档一致：

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

unittest模块不像doctest模块那样容易，不过它可以在一个独立的文件里提供一个更全面的测试集：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

10.12. Batteries Included

Python 有"Batteries Included"的哲学。这最好是通过其较大的文件包的先进和强大功能。例如：

- xmlrpc.client和xmlrpc.server模块让远程过程调用变得轻而易举。尽管模块有这样的名字，它不需要直接XML知识或处理 XML 。

- email包是是一个处理电子邮件的库，包括MIME和其它基于RFC 2822的邮件。与smtplib和poplib用于实际发送和接收邮件，email包有一个完整的工具集用于构建或者解码复杂邮件结构（包括附件），并实现互联网编码和头协议。
- xml.dom和xml.sax的包为这种流行的数据交换格式提供了强大的支持。同样，csv模块支持以常见的数据库格式直接读取和写入。这些模块和包一起大大简化了 Python 应用程序和其他工具之间的数据交换。
- 国际化支持模块包括gettext、locale和codecs包。

11. 标准库概览 — 第II部分

第二部分提供了更高级的模块用来支持专业编程的需要。这些模块很少出现在小型的脚本里。

11.1. 输出格式

reprlib模块提供一个定制版的repr()用于显示大型或者深层嵌套容器：

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

pprint模块提供更复杂的打印控制，以解释器可读的方式打印出内置对象和用户定义的对象。当结果超过一行时，这个"漂亮的打印机"将添加分行符和缩进，以更清楚地显示数据结构：

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap模块格式化文本段落以适应设定的屏宽：

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale模块会访问区域性特定数据格式的数据库。分组属性的区域设置的格式函数的格式设置的数字以直接的方式提供了组分隔符：

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. 模板

string模块包括一个通用Template类，它用简化的语法适合最终用户编辑。这允许用户自定义他们的应用程序无需修改应用程序。

这种格式使用的占位符名称由$与有效的 Python 标识符（字母数字字符和下划线）组成。周围的大括号与占位符允许它应遵循的更多字母数字字母并且中间没有空格。$$创建一个转义的$：

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

当字典或关键字参数中没有提供占位符时，substitute()方法将引发KeyError。对于邮件-合并风格的应用程序，用户提供的数据可能不完整，这时safe_substitute()方法可能会更合适 —— 如果没有数据它将保持占位符不变：

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 类的子类可以指定自定义的分隔符。例如，图像浏览器的批量命名工具可能选用百分号作为表示当前日期、图像 序列号或文件格式的占位符：

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

模板的另一个应用是把多样的输出格式细节从程序逻辑中分类出来。这使它能够替代用户的 XML 文件、 纯文本报告和 HTML 网页报表。

11.3. 二进制数据记录格式

The struct module provides pack() and unpack() functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the zipfile module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4. 多线程

线程是一种解耦非顺序依赖任务的技术。线程可以用来提高接应用程序受用户输入的响应速度，而其他任务同时在后台运行。一个相关的使用场景是 I/O 操作与另一个线程中的计算并行执行。

下面的代码演示在主程序连续运行的同时，threading模块如何在后台运行任务：

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

多线程应用程序的最主要挑战是协调线程间共享的数据或其他资源。为此目的，该线程模块提供了许多同步原语包括锁、 事件、 条件变量和信号量。

尽管这些工具很强大，很小的设计错误也可能导致很难复现的问题。因此，任务协调的首选方法是把对一个资源的所有访问集中在一个单独的线程中，然后使用queue模块用那个线程服务其他线程的请求。应用程序使用Queue对象进行线程间的通信和协调将更容易设计、 更具可读性和更可靠。

11.5. 日志

logging模块提供了一个具有完整功能并且非常灵活的日志系统。最简单的，发送消息到一个文件或者sys.stderr：

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

这将生成以下输出：

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

默认情况下，信息和调试消息被压制并输出到标准错误。其他输出选项包括将消息通过email、 datagrams、sockets发送，或者发送到 HTTP 服务器。根据消息的优先级，新的过滤器可以选择不同的方式：DEBUG、INFO、WARNING、ERROR和CRITICAL。

日志系统可以直接在 Python 代码中定制，也可以不经过应用程序直接在一个用户可编辑的配置文件中加载。

11.6. 弱引用

Python 会自动进行内存管理 (对大多数的对象进行引用计数和垃圾回收以循环利用）。在最后一个引用消失后，内存会立即释放。

这个方式对大多数应用程序工作良好，但是有时候会需要跟踪对象，只要它们还被其它地方所使用。不幸的是，只是跟踪它们也会创建一个引用，这将使它们永久保留。weakref模块提供工具用来无需创建一个引用跟踪对象。当不再需要该对象时，它会自动从 weakref 表中删除并且会为 weakref 对象触发一个回调。典型的应用包括缓存创建的时候需要很大开销的对象：

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python34/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 列表工具

很多数据结构使用内置列表类型就可以满足需求。然而，有时需要其它具有不同性能的替代实现。

The array module provides an array() object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections模块提供了一个deque()对象，就像一个列表,不过它从左边添加和弹出更快，但是在内部查询更慢。这些对象非常实现队列和广度优先的树搜索：

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

除了列表的替代实现，该库还提供了其它工具例如bisect模块中包含处理排好序的列表的函数：

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq模块提供的函数可以实现基于常规列表的堆。最小的值总是保持在第零个位置。这对循环访问最小元素，但是不想运行完整列表排序的应用非常有用：

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8. 十进制浮点数运算

decimal模块提供一个Decimal数据类型用于为十进制浮点运算。相比二进制浮点数内置的float实现，这个类对于以下情形特别有用：

- 财务应用程序和其他用途，需要精确的十进制表示形式，
- 控制精度，
- 对符合法律或法规要求，舍入的控制
- 跟踪有效小数位
- 用户希望计算结果与手工计算相符的应用程序。

例如，计算上 70%电话费的 5%税给不同的十进制浮点和二进制浮点结果。区别变得明显如果结果舍入到最接近的分:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Decimal的结果总是保有结尾的0，自动从两位精度延伸到4位。Decimal 类似手工完成的数学运算，这就避免了二进制浮点数无法精确表达数据精度产生的问题。

精确地表示允许Decimal可以执行二进制浮点数无法进行的模运算和等值测试：

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

decimal模块提供任意精度的运算：

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12.现在怎么办?

阅读本教程可能让你对使用 Python 更感兴趣了——你应该会渴望将 Python 应用于解决实际问题。你应该到哪里去了解更多 Python 的内容呢？

本教程是 Python 文档集的一部分。文档集中的一些其它文件有：

- *Python 标准库*：

你应该浏览本手册，它给出了标准库中关于类型、 函数和模块的完整（虽然简洁）的参考资料。标准的 Python 发布包含大量 的附加模块。其中有读取 Unix 邮箱、收取HTTP 文档、生成随机数、解析命令行选项、编写 CGI 程序、压缩数据以及很多其它任务的模块。浏览一下这个库参考手册会让你知道有什么是现成可用的。

- *安装 Python 模块* 解释如何安装由其他 Python 用户编写的外部模块。
- *Python 语言参考*：详细地讲述了 Python 的语法和语义。它读起来很难，但是作为语言本身的完整指南非常有用。

更多的 Python 资源：

- http://www.python.org：主要的 Python Web 站点。它包含代码、文档和网上 Python 相关页面的链接。该网站在世界各地都有镜像，如欧洲、 日本和澳大利亚；镜像可能会比主站快，这取决于你的地理位置。
- http://docs.python.org：快速访问 Python 的文档。
- http://pypi.python.org： Python 包索引，以前的绰号叫奶酪店，是用户创建的 Python 模块的索引，这些模块可供下载。一旦你开始发布代码，你可以在这里注册你的代码这样其他人可以找到它。
- http://aspn.activestate.com/ASPN/Python/Cookbook/: 这本 Python 食谱收集了相当多的代码示例、大型的模块，以及有用的脚本。其中尤其显著的贡献被收集成一书，这本书也叫做 Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3)。
- http://scipy.org：The Scientific Python 项目包括数组快速计算和处理模块，和大量线性代数、傅里叶变换、非线性solvers、随机数分布，统计分析以及类似的包。

Python 相关的问题和问题报告，你可以发布到新闻组*comp.lang.python*，或将它们发送到邮件列表python-list@python.组织结构图。新闻组和邮件列表是互通的，因此发布到其中的一个消息将自动转发给另外一个。一天大约有 120 个帖子（最高峰到好几百)，包括询问（和回答）问题，建议新的功能和宣布新的模块。在发帖之前，一定要检查常见问题（也称为FAQ）的列表。可在http://mail.python.org/pipermail/查看邮件列表的归档。FAQ 回答了很多经常出现的问题，可能已经包含你的问题的解决方法。

13. 交互式输入的编辑和历史记录

某些版本的 Python 解释器支持编辑当前的输入行和历史记录，类似于在 Korn shell 和 GNU Bash shell 中看到的功能。这是使用GNU Readline库实现的，它支持各种编辑风格。这个库有它自己的文档，在这里我们不就重复了。

13.1. Tab 补全和历史记录

变量和模块名的补全在解释器启动时自动打开以便Tab键调用补全功能；它会查看Python语句的名字，当前局部变量以及可以访问的模块名。对于点分表达式如string.a，它将求出表达式最后一个'.'之前的值，然后根据结果的属性给出补全的建议。注意，如果一个具有**getattr()**方法的对象是表达式的某部分，这可能执行应用程序定义的代码。默认的配置同时会把历史记录保存在你的用户目录下一个名为.python_history的文件中。在下次与交互式解释器的回话中，历史记录将还可以访问。

14. 浮点数运算：问题和局限

浮点数在计算机硬件中表示为以 2 为底（二进制）的小数。例如，十进制小数

```
0.125
```

是1/10 + 2/100 + 5/1000 的值，同样二进制小数

```
0.001
```

是 0/2 + 0/4 + 1/8 的值。这两个小数具有相同的值，唯一真正的区别是，第一个小数是十进制表示法，第二个是二进制表示法。

不幸的是，大多数十进制小数不能完全用二进制小数表示。结果是，一般情况下，你输入的十进制浮点数仅由实际存储在计算机中的近似的二进制浮点数表示。

这个问题在十进制情况下很容易理解。考虑分数 1/3，你可以用十进制小数近似它：

```
0.3
```

或者更接近的

```
0.33
```

或者再接近一点的

```
0.333
```

等等。无论你愿意写多少位数字，结果永远不会是精确的 1/3，但将会越来越好地逼近 1/3。

同样地，无论你使用多少位的二进制数，都无法确切地表示十进制值 0.1。1/10 用二进制表示是一个无限循环的小数。

```
0.0001100110011001100110011001100110011001100110011...
```

在任何有限数量的位停下来，你得到的都是近似值。今天在大多数机器上，浮点数的近似使用的小数以最高的53位为分子，2的幂为分母。至于1/10这种情况，其二进制小数是 3602879701896397/2**55，它非常接近但不完全等于1/10真实的值。

由于显示方式的原因，许多使用者意识不到是近似值。Python 只打印机器中存储的二进制值的十进制近似值。在大多数机器上，如果 Python 要打印 0.1 存储的二进制的真正近似值，将会显示

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这么多位的数字对大多数人是没有用的，所以 Python 显示一个舍入的值

```
>>> 1 / 10
0.1
```

只要记住即使打印的结果看上去是精确的1/10，真正存储的值是最近似的二进制小数。

例如，数字0.1、0.10000000000000001 和 0.1000000000000000055511151231257827021181583404541015625 都以 3602879701896397/2**55为近似值。因为所有这些十进制数共享相同的近似值，在保持恒等式eval(repr(x))==x的同时，显示的可能是它们中的任何一个。

现在从Python 3.1 开始，Python（在大多数系统上）能够从这些数字当中选择最短的一个并简单地显示0.1。

注意，这是二进制浮点数的自然性质：它不是 Python 中的一个 bug，也不是你的代码中的 bug。你会看到所有支持硬件浮点数算法的语言都会有这个现象（尽管有些语言默认情况下或者在所有输出模式下可能不会显示 出差异）。

为了输出更好看，你可能想用字符串格式化来生成固定位数的有效数字：

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

认识到这，在真正意义上，是一种错觉是很重要的：你在简单地舍入真实机器值的显示。

例如，既然0.1不是精确的1/10，3个0.1的值相加可能也不会得到精确的0.3：

```
>>> .1 + .1 + .1 == .3
False
```

另外，既然0.1不能更接近1/10的精确值而且0.3不能更接近3/10的精确值，使用round()函数提前舍入也没有帮助：

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

虽然这些数字不可能再更接近它们想要的精确值，round()函数可以用于在计算之后进行舍入，这样的话不精确的结果就可以和另外一个相比较了：

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

二进制浮点数计算有很多这样意想不到的结果。“0.1”的问题在下面"误差的表示"一节中有准确详细的解释。更完整的常见怪异现象请参见浮点数的危险。

最后我要说，“没有简单的答案”。也不要过分小心浮点数！Python 浮点数计算中的误差源之于浮点数硬件，大多数机器上每次计算误差不超过 2**53 分之一。对于大多数任务这已经足够了，但是你要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的舍入错误。

虽然确实有问题存在，对于大多数平常的浮点数运算，你只要简单地将最终显示的结果舍入到你期望的十进制位数，你就会得到你期望的最终结果。str()通常已经足够用了，对于更好的控制可以参阅格式化字符串语法中str.format()方法的格式说明符。

对于需要精确十进制表示的情况，可以尝试使用decimal模块，它实现的十进制运算适合会计方面的应用和高精度要求的应用。

fractions模块支持另外一种形式的运算，它实现的运算基于有理数（因此像1/3这样的数字可以精确地表示）。

如果你是浮点数操作的重度使用者，你应该看一下由SciPy项目提供的Numerical Python包和其它用于数学和统计学的包。参看http://scipy.org。

当你真的真想要知道浮点数精确值的时候，Python 提供这样的工具可以帮助你。float.as_integer_ratio()方法以分数的形式表示一个浮点数的值：

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

因为比值是精确的，它可以用来无损地重新生成初始值：

```
>>> x == 3537115888337719 / 1125899906842624
True
```

float.hex()方法以十六进制表示浮点数，给出的同样是计算机存储的精确值：

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

精确的十六进制表示可以用来准确地重新构建浮点数：

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

因为可以精确表示，所以可以用在不同版本的Python（与平台相关）之间可靠地移植数据以及与支持同样格式的其它语言（例如Java和C99）交换数据。

另外一个有用的工具是math.fsum()函数，它帮助求和过程中减少精度的损失。当数值在不停地相加的时候，它会跟踪“丢弃的数字”。这可以给总体的准确度带来不同，以至于错误不会累积到影响最终结果的点：

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

14.1. 二进制表示的误差

这一节将详细解释“0.1”那个示例，并向你演示对于类似的情况自已如何做一个精确的分析。假设你已经基本了解浮点数的二进制表示。

二进制表示的误差 指的是这一事实，一些（实际上是大多数) 十进制小数不能精确地用二进制小数表示。这是为什么 Python（或者 Perl、 C、 C++、 Java、 Fortran和其他许多语言）通常不会显示你期望的精确的十进制数的主要原因。

这是为什么？1/10 和 2/10 不能用二进制小数精确表示。今天（2010 年 7 月）几乎所有的机器都使用 IEEE-754 浮点数算法，几乎所有的平台都将 Python 的浮点数映射成 IEEE-754“双精度浮点数”。754 双精度浮点数包含 53 位的精度，所以输入时计算机努力将 0.1 转换为最接近的 *J*/2***N* 形式的小数，其中*J* 是一个 53 位的整数。改写

```
1 / 10 ~= J / (2**N)
```

为

```
J ~= 2**N / 10
```

回想一下 *J* 有 53 位（>=2**52**但**<2**53），所以 *N* 的最佳值是 56：

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

即 56 是 *N* 保证 *J* 具有 53 位精度的唯一可能的值。*J* 可能的最佳值是商的舍入：

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

由于余数大于 10 的一半，最佳的近似值是向上舍入：

```
>>> q+1
7205759403792794
```

因此在 754 双精度下 1/10 的最佳近似是：

```
7205759403792794 / 2 ** 56
```

分子和分母都除以2将小数缩小到：

```
3602879701896397 / 2 ** 55
```

请注意由于我们向上舍入，这其实有点大于 1/10；如果我们没有向上舍入，商数就会有点小于 1/10。但在任何情况下它都不可能是精确的 1/10!

所以计算机从来没有"看到"1/10： 它看到的是上面给出的精确的小数，754 双精度下可以获得的最佳的近似了：

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

如果我们把这小数乘以 10**55，我们可以看到其55位十进制数的值：

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

也就是说存储在计算机中的精确数字等于十进制值
0.1000000000000000055511151231257827021181583404541015625。许多语言（包括旧版本的Python）会把结果舍入到17位有效数字，而不是显示全部的十进制值：

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

fractions 和 decimal 模块使得这些计算很简单：

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```